

AFRL-IF-RS-TR-2000-137
Final Technical Report
September 2000



MODULAR CONSTRUCTION OF VERY LARGE KNOWLEDGE BASES

David Espinosa

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F102

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

DTIC QUALITY INSPECTED 4
20001222 108

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

MODULAR CONSTRUCTION OF VERY LARGE KNOWLEDGE BASES

David Espinosa

Contractor: Kestral Institute

Contract Number: F30602-97-C-0146

Effective Date of Contract: 30 May 1997

Contract Expiration Date: 30 September 1999

Short Title of Work: Modular Construction of Very Large Knowledge Bases

Period of Work Covered: May 97 - Sep 99

Principal Investigator: David Espinosa

Phone: (415) 493-6871

AFRL Project Engineer: Craig S. Anken

Phone: (315) 330-2074

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Craig S. Anken, AFRL/IFTD, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
<p>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</p>			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED	
	SEPTEMBER 2000	Final May 97 - Sep 99	
4. TITLE AND SUBTITLE	MODULAR CONSTRUCTION OF VERY LARGE KNOWLEDGE BASES		
6. AUTHOR(S)	David Espinosa		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	Kestrel Institute 3260 Hillview Avenue Palo Alto CA 04304		
8. PERFORMING ORGANIZATION REPORT NUMBER	N/A		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	Defense Advanced Research Projects Agency Air Force Research Laboratory/IFTD 3701 North Fairfax Drive 525 Brooks Road Arlington VA 22203-1714 Rome NY 13441-4505		
10. SPONSORING/MONITORING AGENCY REPORT NUMBER	AFRL-IF-RS-TR-2000-137		
11. SUPPLEMENTARY NOTES	Air Force Research Laboratory Project Engineer: Craig S. Anken/IFTD/(315) 330-2074		
12a. DISTRIBUTION AVAILABILITY STATEMENT	APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.		
12b. DISTRIBUTION CODE			
13. ABSTRACT (Maximum 200 words) In this report, we describe work done at Kestrel under the DARPA High Performance Knowledge Bases program.. The goal of the program is to develop methods for structuring large knowledge-bases and reasoning efficiently in them. The report contains four sections. In the first, we describe our work on the crisis management challenge problems. In the second, we describe our Designware system for semi-automated program synthesis. In the third, we present a detailed description of parametrized specifications, an important tool for combining theory refinements. In the fourth, we describe limits, interpretation, and slicing, all important tools for constructing, deconstructing, and relating theories.			
14. SUBJECT TERMS	Knowledge-Bases, Formal Methods, Reasoning and Logic		15. NUMBER OF PAGES 72
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Table of Contents

1. Introduction	4
2. Challenge problem experiments	4
2.1. Conventional approach	6
2.1.1.Escalation and retaliation	6
2.1.2.Hostility levels	7
2.1.3.Procedural attachment	8
2.1.4.Axiomatic Reasoning	9
2.1.5.Example	11
2.2. Finite models	12
2.3. Parametric queries	13
2.4. Conditional formation	14
2.5. Temporal reasoner	15
2.5.1.Table generation with Specware	16
3. Designware	16
3.1. Overview	17
3.2. Basic Concepts	18
3.2.1.Specifications	18
3.2.2.Morphisms	19
3.2.3.The Category of Specs	21
3.2.4.Diagrams	22
3.2.5.The Structuring of Specifications	23
3.2.6.Refinement and Diagrams	23
3.2.7.Logic Morphisms and Code Generation	24
3.3. Software Development by Refinement	25
3.3.1.Constructing Specifications	25
3.3.2.Constructing Refinements	25
3.4. Scaling up	26
3.4.1.Design by Classification: Taxonomies of Refinements	27
3.4.2.Tactics	29
3.5. Summary	30
4. Parametrized specifications	30
4.1. Introduction	31
4.1.1.Parametric specifications	31
4.1.2.Elements of categorical model theory	32
4.1.3.Outline of the paper	33
4.2. Theories and models, categorically	34

4.2.1. Classifying categories	34
4.2.2. Coherent categories	34
4.2.3. Interpretations and models	36
4.2.4. Parametrized specifications as functors: syntactic vs semantic definitions	36
4.3. Syntactic vs semantic properties of functors	38
4.3.1. Preliminaries	38
4.3.2. Basic results	39
4.4. Characterizing parametric specifications	42
4.5. Conclusions and further work	47
5. Other mathematical methods	48
5.1. Limits	48
5.2. Interpretations	53
5.3. Slicing	55

1. Introduction

In this report, we describe work done at Kestrel under the DARPA High Performance Knowledge Bases program. The goal of the program is to develop methods for structuring large knowledge bases and reasoning efficiently in them.

The report contains four sections. In the first, we describe our work on the crisis management challenge problems. In the second, we describe our Designware system for semi-automated program synthesis. In the third, we present a detailed description of parametrized specifications, an important tool for combining theory refinements. In the fourth, we describe limits, interpretations, and slicing, all important tools for constructing, deconstructing, and relating theories.

2. Challenge problem experiments

Kestrel applied its work on the technology base for HPKB to address higher performance aspects of the program's goals. Our objective was to show how the tech base underlying technology could provide meaningful speedups on large problems, and to show potential for speedup on even larger problems. We are encouraged by our findings so far: automatically-generated solvers produced speedups ranging from two to five orders of magnitude, with potential applicability also to large KB *acquisition* problems.

SRI graciously suggested challenge questions and furnished Kestrel with base axiom sets and scenario axiom sets, and helped with our analyses. We measured baseline performance and three knowledge compilation approaches:

Method / TFQ	236b	236c	236d	236e	210a	210b	210c
1. Conventional approach (answer extraction)	200 s	199 s	204 s	181 s	94 s	83 s	82 s
2. Knowledge compilation (finite models)	39 ms (5000x)			39 ms (4600x)	2.3 ms (40000x)	3.4 ms (24000x)	3.2 ms (25000x)
3. Knowledge compilation (schemas)	310 ms (600x)	330 ms (600x)	310 ms (600x)	310 ms (600x)			

All times were measured on a Sun Ultrasparc 60.

Each table row shows the time for a different approach:

Conventional approach The conventional approach is answer extraction via tracing of unification substitution (“answer literals”), which SRI used for the TFQs. This technique is due to Green [16]. For this baseline, we used SRI’s SNARK, a good, well-suited, fast prover.

Knowledge Compilation via Finite Models Typically, major portions of a knowledge base have a finite model. For these portions, the model can be pre-computed from the axiom set and incorporated into the prover via procedural attachment to predicates. We used a simple forward closure to compute the model as a set of ground literals. Another point of view is that we used partial evaluation, holding the axiom base fixed. Question answering is then done via simple operations on sets and relations. These operations could be further optimized using bit vectors, and we estimate that another two orders of magnitude should be achievable.

Knowledge Compilation via Schemata Several hundred of the axioms in the HPKB crisis management database have a similar form, varying only by the types of agents, actions, and interests involved. Because an axiom can only mention specific, concrete types, we cannot write a single axiom that generalizes all of them; however, we can summarize them in a table. Then, instead of using a prover for queries, we use table lookup, which is much faster. We could integrate this technique into a general prover for use on suitable subproblems.

Knowledge Compilation via Conditional Compilation (not shown) By working with abstract rather than concrete data, a prover can be outfitted to construct a program that computes a result later, when the actual concrete data is given to it. This principle is also used in partial evaluation. The program typically includes conditional branches that depend on the actual concrete data values, hence the name “conditional compilation”. This technique is also due to Green [16].

We then optimize the resulting program using Kestrel’s C code generator. The resulting program is eight *orders of magnitude* faster than standard answer extraction and produces answers in microseconds rather than seconds. Although this result is surprising, it was done on too small a problem (several hundred clauses) to be anything but suggestive. Extending this approach would require additional research on problem solving via program synthesis.

In our work, we began with SRI’s answers, and, instead of trying to improve them, we focused on knowledge compilation to produce them more quickly. Also, we have been producing answers without explanations, but we expect it would not be difficult to generate the same explanations as a standard prover.

Scalability We expect our knowledge compilation techniques to scale, relative to “pure” inference provers. That is, the compilation techniques should maintain a significant advantage as question difficulty and knowledge base size increase. This is no surprise; special problem solvers are often used to assist provers. However, in our case, the special problem solvers were automatically generated.

We also believe that knowledge compilation can lead to better answers. Although answering (search) speed is not an evaluated property for HPKB, search speed translates into more efficient searches and therefore into larger spaces searched. Larger searches translate into ability to answer harder questions. To pursue this issue, it is possible to *measure* the increase the size of the explored space as a function of degree of compilation for typical questions.

Knowledge acquisition The acquisition of useful, consistent knowledge bases is a difficult problem. Fast query responses help to discover omissions and inconsistencies, but more systematic analysis techniques are needed. Knowledge compilation will allow more analyses, some fast enough to run in the background as new knowledge is entered.

We have not yet developed industrial-strength knowledge compilation tools, although it appears feasible. Knowledge compilation will be essential for answering computationally intensive queries such as the network intervention and repair problems for which we generated algorithms earlier in the HPKB program. Large KBs will also profit from the modularization and composition mechanisms available in Specware.

2.1. Conventional approach

In this section, we describe our first approach to the HPKB Crisis Management challenge problems. This work is unusual because it combine deductive and procedural reasoning using the procedural attachment mechanism of the SNARK theorem prover. In the competition, the solutions we obtain to this class of problems were markedly superior to those of the competing team.

2.1.1. Escalation and retaliation Parametrized question 210 was unusual because, in addition to ordinary axiomatic reasoning, it required the use of a procedural attachment to get some of the effects of nonmonotonic reasoning. Instances of the question require us to determine whether a particular action in a particular context constitutes an escalation, a de-escalation, or a retaliation.

The contexts include both historical incidents and fictional scenarios for the Crisis Management Challenge Problem (CMCP). We regard an action in a context as an escalation if it is in response to another, less hostile action. Conversely, an action is a de-escalation if it is in response to a more hostile action. Finally, an action is a retaliation for another if it is a reaction to the other and the two actions are opposed to each other or have contrary interests.

The formal encodings of the incidents and scenarios contain a detailed account of the agents that each action opposes, the interests that motivate them, and the causal

relationship between actions; this allows us to deal with questions about retaliation. To solve questions involving escalation, however, it is necessary to estimate the level of hostility of each action.

2.1.2. Hostility levels Because there are infinitely many possible individual actions, we cannot assign hostility levels to actions individually. Instead, we compute the hostility level of an action according to its sort and other characteristics, because there is a manageable number of these characteristics, and because they are known in advance.

One notion of hostility level was proposed by Herman Kahn (“On Escalation”), who suggested a forty-four-stage linear scale. This idea was modified by a CMCP subject-matter expert during an interview (“Knowledge Acquisition for Crisis Management: Interests and Actions,” John Kingston, AIAI, University of Edinburgh). He suggested a multi-dimensional scale for hostility levels, because of the difficulty in comparing actions of different kinds. Our own scale currently has three components: damage level, weapon level, and proximity level.

The damage level is an estimate of what kind of damage the action involves. A military attack, for example, is likely to involve population damage, which is the most severe level. Public criticism of one government by another is likely to reach only the verbal damage level, which is much less severe.

The weapon level reflects what sort of weapons the attack involves. For example, an attack using biological weapons has **wmd-level**, the most severe, because it uses a weapon of mass destruction (WMD). An attack that involves no weapons has the lowest weapon level of all.

The proximity level concerns the location of the attack. An attack on the heart of another country has the most severe proximity level – an attack that is outside the borders of the target country has the lowest proximity level.

All the actions in the scenarios and historical narratives are classified according to their sort in the ontology. For instance, in the 1984-88 Tanker War historical account, the invasion of Iran by Iraq is classified as a **military-invasion**; the action in which Kuwait negotiates with other countries to protect its shipping is classified as **making-an-agreement**.

For certain sorts, we provide a damage level and a weapon level. The proximity level of an action is determined not by a sort but by the location of the attack.

For instance, if an action is of sort **terrorist-attack**, it is assigned a **population-damage-level** and a **target-indiscriminate-conventional-level** weapon level. If it is performed in the capital city of the target country, it is of **heart-proximity-level**, the most severe.

Hostility levels of two actions can be compared using a lexicographic ordering. More precisely, the action with the higher damage level has the higher hostility level;

if the damage levels are equal, the action with the higher weapon level has the higher hostility level; and if both the damage levels and the weapon levels are equal, the action with the higher proximity level has the higher hostility level.

For example, an attack that kills population is regarded as more severe than one that only damages property, even if the first uses pistols (*target-specific-weapon-level*) and is in on the outskirts of the target country (*inside-proximity-level*), while the second uses a bomb (*target-indiscriminate-conventional-level*) and is in the center of the target country (*heart-proximity-level*).

2.1.3. Procedural attachment It was impossible to use axioms in the ordinary way to define hostility levels of actions, because of the nonmonotonicity of the problem. It is quite usual for an action to be of more than one sort; for example, one sort may be a subsort of another, so an action of the first sort is also of the second sort. If we use assignment axioms of the form

```
if ?action is of sort Ai  
then damage-level(?action) = di
```

and an action is of two sorts A1 and A2, it might be assigned conflicting damage levels d1 and d2, leading to an inconsistent axiom set.

For instance, one sort in the ontology, *convene-task-force-to-monitor-responses*, is assigned a de-escalation damage level, which is very low, because it is involved in truce making. This sort, however, is classified in the ontology as a subsort of a larger sort, *political action*. Actions of this sort are assigned a verbal damage level, which is somewhat more severe, because hostile political actions such as threat-making are also of this sort.

The situation is illustrated by this table:

Sort	Damage level
<i>convene-task-force-to-monitor-responses</i>	<i>de-escalation-damage-level</i>
<i>political-action</i>	<i>verbal-damage-level</i>

At one stage in the CMCP Year 2 Scenario, the UN Secretary General persuades Iran and the GCC to participate in a Persian Gulf regional forum. This action is classified in the scenario description as being of sort *convene-task-force-to-monitor-responses*. Hence, the above assignment axiom can be applied to assign the action a de-escalation damage level. Since the agreement of Iran and the GCC is also a political action, the assignment axiom could be applied as well to assign it a verbal damage level, contradicting the previous assignment.

We would prefer to compute the damage level of an action by using the smallest subsort that includes the action and that has an assigned damage level, since the

smaller subsorts give more specific information. If a subsort has no assigned damage level, we use the damage level of larger subsorts.

Such an operation is not monotonic: finding out new information, such as an assignment of a damage level to a sort, could result in an inference becoming invalid. For example, if an inference depended on the fact that the agreement has a verbal damage level, and then we discover later that convening a task force has a lower damage level, the earlier inference will no longer be valid.

Axiomatic reasoning can perform only monotonic inferences, which does not allow the retraction of any conclusion as a result of discovering new evidence. Therefore, instead of using axiomatic reasoning, we invoke a procedural attachment mechanism to compute damage and weapon levels.

Procedural attachment is a SNARK feature that enables us to associate selected predicate and function symbols in SNARK with Lisp functions that evaluate them. This allows us to circumvent axiomatic reasoning when we have a procedural way of determining the value of the symbol. In this case, we have procedural attachments for the `damage-level` and `weapon-level` function symbols. In computing the level for an action, the attachment looks at the sort(s) of that action. The damage and weapon levels of the smallest sort with an assigned level is assigned as the damage and weapon levels of the action. If there is more than one such smallest sort, the highest of the levels is selected.

For example, in determining the damage level of the action in which Iran and the GCC make the agreement, the function will assign the de-escalation damage level, which is associated with the sort `convene-task-force-to-monitor-responses`. The verbal damage level, which is associated with the sort `political-action`, is ignored, because `convene-task-force-to-monitor-responses` is a proper subsort of `political-action` – it is smaller.

2.1.4. Axiomatic Reasoning Although procedural attachment is used to compute hostility levels, ordinary axiomatic reasoning is used to determine whether an action in a narrative is to be regarded as an escalation, a de-escalation, or a retaliation. For instance, one axiom states that an action is an escalation if it is caused by another action and is of a greater hostility level than the earlier action. More precisely, the escalation axiom is

```
(assertion
(<=
(escalation ?action2 ?context)
(and
(occurs-in ?action2 ?context)
(cause-event-event* ?action1 ?action2))
```

```

(greater-hostility ?action2 ?action1)))
:name escalation-if-in-response-to-less-hostile-action
:documentation "An ?action2 is an escalation in ?context
if it is in response to an ?action1 and is of greater hostility
level than ?action1.")

```

A similar axiom for de-escalation is

```

(assertion
(<=
(de-escalation ?action2 ?context)
(and
(occurs-in ?action1 ?context)
(cause-event-event* ?action1 ?action2)
(greater-hostility ?action1 ?action2)))
:name de-escalation-if-in-response-to-more-hostile-action
:documentation "An ?action2 is a de-escalation in ?context
if it is in response to an ?action1 that has a greater
hostility level")

```

There are other axioms for escalation and de-escalation too. For instance, an action whose damage level is de-escalation (for example, a ceasefire) is automatically regarded as a de-escalation.

An action is a retaliation for another action if it is caused by that action and the two actions are opposed to each others agents or are motivated by opposing interests. For instance, if ?agent1 performs an action that damages the interests of ?agent2, and in response ?agent2 performs an action that damages the interests of ?agent1, the latter action is presumed to be a retaliation. More precisely, we have the retaliation axiom

```

(assertion
(<=
(retaliation ?action2 ?action1 ?context)
(and
(occurs-in ?action2 ?context)
(performed-by ?action2 ?agent2)
(performed-by ?action1 ?agent1)
(cause-event-event* ?action1 ?action2)
(or
(and (opposing ?action1 ?agent2)
(opposing ?action2 ?agent1)))

```

```

(and (opposes-interest ?action1 ?interest1)
     (supports-interest ?action2 ?interest1))
(and (supports-interest ?action1 ?interest2)
     (opposes-interest ?action2 ?interest2))))
:name retaliation-is-in-response-to-earlier-action
:documentation "A retaliation is an action caused by an earlier action,
such that the actions are opposed to each others agents or
the two actions have opposing interests.")

```

2.1.5. Example Let us see how these techniques applied to a sample Challenge Problem. In the final evaluation, question TFQ210c asks: "In the 1984-8 Tanker War, is the event in which Iraq accede to request of UN during 20 July 1987 a de-escalation of conflict?" Note that the question is ungrammatical because it is generated automatically from a template.

This question is formalized as follows:

```

(and (occurs-in ?action2 1984-1988-tanker-war)
     (instance-of ?action2 accept)
     (performed-by ?action2 iraq)
     (action-involves ?action2 ?agreement)
     (action-enabled-by ?action2 ?action1)
     (performed-by ?action1 united-nations)
     (instance-of ?action1 making-an-agreement)
     (temporal-bounds-contain
      (day-fn 20 (month-fn july (year-fn 1987))) ?action1)
     (de-escalation ?action2 1984-1988-tanker-war)))

```

Most of the formalization simply describes the event, paraphrasing the question. This serves to bind to ?action2 the event under discussion. The key component of the formalization is the final conjunct,

```
(de-escalation ?action2 1984-1988-tanker-war),
```

which asks if the action is a de-escalation.

By the axiom for de-escalation we gave earlier, the system knows that an action is a de-escalation if it is in response to an earlier action of greater hostility. In the formal description of the Tanker War, it discovers an axiom

```

(contributing-factor
  iran-directly-challenges-ships-in-us-convoys
  iraq-accepts-un-resolution-598)

```

In other words, a contributing factor in Iraq's acceptance of the UN resolution is the fact that Iran has directly challenged ships in a US convoy.

An axiom in the knowledge base tells us that if one action is a contributing factor in another, it is a cause of the other action:

```
(assertion
  (<=
    (cause-event-event* ?event1 ?event2)
    (contributing-factor ?event1 ?event2))
  :name contributing-factor-implies-causes-indirectly
  :documentation "If an event is a contributing factor for another,
  it also causes it indirectly" )
```

Therefore, Iran's challenge of the US convoy is a cause of Iraq's acceptance of the UN resolution.

Acceptance of the UN resolution is declared in the formalization of the Tanker War to be of sort **making-an-agreement**, which has a verbal damage level, which is relatively low. On the other hand, Iran's challenge of the US convoy is declared to be of sort **threatening-action**, which is declared to have damage level **non-combat-action**. Since this damage level is higher than that of a verbal action, the hostility level of Iran's challenge is higher than that of Iraq's acceptance of the UN resolution. This is because the ordering on hostility levels has been asserted to be a lexicographic ordering on its components.

In short, Iraq's acceptance of the UN resolution has been found to be in response to another action of higher hostility. Hence, by the de-escalation axiom, it is a de-escalation.

2.2. Finite models

In this approach, we observe that major portions of a KB have a finite model, and that we can reason more quickly in this model than in the KB itself.

To build the model, we compute the transitive closure of the axioms, and attach to each predicate the set of its instances. Then, to answer a query, we find the instances satisfying each literal and combine them using intersection and union.

Although computing the transitive closure took some time, the resulting ground unit KB was quite manageable in size. Query answering took less than a second, and aggressive optimization probably could reduce the response time to under one millisecond.

This technique works directly from a given KB, with little or no human intervention. Although preprocessing is time-consuming, much of this work can be avoided if the original KB is organized in tables of ground data.

2.3. Parametric queries

For parametric queries, such as PQ210, we found that axioms and explanations came in symmetry groups. A general query asked once and for all could be reused by instantiation to all members in a given symmetry group. For PQ210, we were able to group over 200 axioms, thus representing about 1000 different explanations within a single scheme.

Structuring a set of axioms in this way helped to identify missing axioms. Under the sort assumption that a terrorist group is an instance of a criminal organization we found that our added axioms could answer all queries under TQ210.

Knowledge is captured using generic templates that are summarized in tables instead of enumerating very similar axioms in a verbose mode.

For the first, and largest template

(prove-performed-meta-theorem)

proves the generic theorem that justifies retrieving answers by a simple table lookup instead of querying the knowledge base.

(evaluate-interest-typically-underlies agent action)

performs this lookup, whereas

(ask-interest-typically-underlies agent action)

does the corresponding theorem proving directly. However, given the meta-proof, we obtain an explanation for each instance by simple substitution of the parameter agent and action sorts into the explanation (which we cannot access yet as we need a compatible version of OKBC and other SRI/SAIC installed layers on top of SNARK and LISP).

The timing effect on precompiling a generic query answer function for the template class of axioms which constituted about 90 percent of the available data is dramatic. To answer, for instance, what the interests that typically underly a country to conduct a peace keeping mission SNARK generates the last answer instance at the 7,101 derived clause:

```
; Summary of computation:  
; 7,101 formulas have been input or derived (from 5,454 formulas).  
; 6,609 (93%) were retained. Of these,  
; 793 (12%) were simplified or subsumed later,  
; 0 ( 0%) were deleted later because the agenda was full, and  
; 5,816 (88%) are still being kept.
```

```

;
; Run time in seconds excluding printing time:
;      1.5   2% Resolution
;      0.5   0% Paramodulation
;     14.0  14% Forward subsumption
;      2.1   2% Backward subsumption
;      2.2   2% Forward simplification
;      0.7   1% Backward simplification
;     58.5  60% Sorts
;      9.9  10% Kif input
;      8.8   9% Other
;     98.2    Total

```

The alternative available answering mechanism that uses a simple table lookup is in contrast instantaneous.

From a meta theorem, we can generate the table by extracting all model axioms as answers and have the query answering mechanism dispatch on the sorts in the table entries. The table can naturally also be generated a priori, because the axioms reflect the nature of the table.

2.4. Conditional formation

In this approach, we convert axioms into conditional expressions, then use these expressions to create a highly optimized C program. For example, if we begin with the axiom

```
nation(x) iff x = USA or x = England or ...
```

we generate this program

```
boolean nation (object x) {
  (x == USA) || 
  (x == England) || ...
}
```

Using this method, we obtained dramatic speedups (answers in microseconds), but we only tried small examples. Still, potentially enormous improvements in performance are plausible.

2.5. Temporal reasoner

Reasoning about events in time is pervasive in the HPKB knowledge base. The temporal logic covered by the Allen temporal relations extended with date reasoning has been adequate to capture most of the relevant properties.

In part of Kestrel's collaboration with Cycorp we developed a temporal reasoner based on the Allen temporal relations in Cycorp's knowledge base programming language μ LISP. The temporal reasoner maintains a data base of relations between time intervals. Intervals I and J can be related in the following ways:

1. $\text{before}(I, J)$: I is strictly before J .
2. $\text{meets}(I, J)$: I ends where J starts.
3. $\text{join}(I, J)$: I starts strictly before J and ends inside J .
4. $\text{??}(I, J)$: I starts strictly before J and ends where J ends.
5. $\text{??}(I, J)$: I starts where J starts and ends strictly before J ends.
6. $\text{equal}(I, J)$: intervals I and J are the same.

together with the symmetric versions, for example $\text{met-by}(I, J) \leftrightarrow \text{meet}(J, I)$.

Since all the possible relations between two closed intervals are enumerated by this base set of relations it is directly closed under negation. The negation of a temporal relation R_i is the disjunction of the other relations $\bigvee_i R_i$. The temporal reasoner uses a 7 by 7 table to compute the transitive closure of relations. For example, if $\text{before}(I, J)$ holds and $\text{join}(J, K)$, then $\text{before}(I, K)$. On the other hand, if $\text{meets}(I, J)$ and $\text{meets}(J, K)$, then either $\text{meets}(I, K)$ or $\text{before}(I, K)$. The table summarizing all Allen temporal relationships is summarized in [3]. However, it contains a typo, a disjunction in one of the table entries is missing, so to avoid repeating the same typo and introducing new we synthesized the table using decision procedures for rational numbers. For this purpose we translated the Allen temporal primitives to arithmetical relations using the schema:

$$\begin{aligned} \text{before}(I, J) &= \text{End}(I) < \text{Start}(J) \\ \text{meets}(I, J) &= \text{End}(I) = \text{Start}(J) \\ \text{starts}(I, J) &= \text{Start}(I) = \text{Start}(J) \& \text{End}(I) < \text{End}(J) \\ &\dots \end{aligned}$$

Then, for I, J, K we formed all triples of the form

$$R_i(I, J) \wedge R_j(J, K) \wedge R_k(I, K)$$

for temporal relations R_i, R_j, R_k , and included R_k in the R_i, R_j entry if and only if the arithmetical predicate was satisfiable (the fact that we used a decision procedure for rational arithmetic was not exploited fully, as the decision problems only required reasoning about a linear order).

Using this approach we generated the relevant table for the Allen temporal reasoner and delivered it to Cycorp.

2.5.1. Table generation with Specware In this paragraph we outline how the table was generated using Specware's user syntax.

```

spec IntervalRelations =
  sort interval
  op Start : interval -> Nat
  op End   : interval -> Nat

  axiom fa(i:interval) (Start(i) <= End(i))

  def Before(I,J) = End(I) < Start(J)
  def After(I,J) = Before(J,I)
  def During(I,J) =
    Start(J) < Start(I) & End(I) < End(J)
  def Contains(I,J) = During(y,I)
  def Overlaps(I,J) =
    Start(I) < Start(J) &
    Start(J) < End(I)   &
    End(I) < End(J)
  def OverlappedBy(I,J) = Overlaps(y,I)
  def Meets(I,J) = End(I) = Start(J)
  def MetBy(I,J) = Meets(y,I)
  def Starts(I,J) =
    Start(I) = Start(J) & End(I) < End(J)
  def StartedBy(I,J) =
    Starts(y,I)
  def Finishes(I,J) =
    End(I) = End(J) & Start(J) < Start(I)
  def FinishedBy(I,J) = Finishes(J,I)

  conjecture *tr-lt*-1 is
    not(trStart(a,b) & trStart(b,c) & trStart(a,c))
  ...
  conjecture *tr-fc*-49 is
    not(trFinshedBy(a,b) & trFinishedBy(b,c) & trFinishedBy(a,c))
end-spec

```

Curiously, we later found that the above mentioned bug in the article had penetrated to other implementations of the Allen temporal reasoner, which ended up giving undesired behaviour of the underlying theorem prover.

3. Designware

This section presents a mechanizable framework for software development by refinement. The framework is based on a category of higher-order specifications. The key idea is representing knowledge about programming concepts, such as algorithm design, datatype refinement, and expression simplification, by means of taxonomies of specifications and morphisms.

The framework is partially implemented in the research systems Specware, Designware, and Planware. Specware provides basic support for composing specifications and refinements via colimit, and for generating code via logic morphisms. Specware is intended to be general-purpose and has found use in industrial settings. Designware extends Specware with taxonomies of software design theories and support for constructing refinements from them. Planware builds on Designware to provide highly automated support for requirements acquisition and synthesis of high-performance scheduling algorithms.

3.1. Overview

A software system can be viewed as a composition of information from a variety of sources, including

- the application domain,
- the requirements on the system’s behavior,
- software design knowledge about system architectures, algorithms, data structures, code optimization techniques, and
- the run-time hardware/software/physical environment in which the software will execute.

This section presents a mechanizable framework for representing these various sources of information, and for composing them in the context of a refinement process. The framework is founded on a category of specifications. Morphisms are used to structure and parameterize specifications, and to refine them. Colimits are used to compose specifications. Diagrams are used to express the structure of large specifications, the refinement of specifications to code, and the application of design knowledge to a specification.

The framework features a collection of techniques for constructing refinements based on formal representations of programming knowledge. Abstract algorithmic concepts, datatype refinements, program optimization rules, software architectures, abstract user interfaces, and so on, are represented as diagrams of specifications and morphisms. We arrange these diagrams into taxonomies, which allow incremental access to and construction of refinements for particular requirement specifications. For example, a user may specify a scheduling problem and select a theory of global search algorithms from an algorithm library. The global search theory is used to construct a refinement of the scheduling problem specification into a specification containing a global search algorithm for the particular scheduling problem.

The framework is partially implemented in the research systems Specware, Designware, and Planware. Specware provides basic support for composing specifications and refinements, and generating code. Code generation in Specware is supported by

inter-logic morphisms that translate between the specification language/logic and the logic of a particular programming language (e.g. CommonLisp or C++). Specware is intended to be general-purpose and has found use in industrial settings. Designware extends Specware with taxonomies of software design theories and support for constructing refinements from them. Planware provides highly automated support for requirements acquisition and synthesis of high-performance scheduling algorithms.

The remainder of this section covers basic concepts and the key ideas of our approach to software development by refinement, in particular the concept of design by classification [26]. We also discuss the application of these techniques to domain-specific refinement in Planware [5]. A detailed presentation of a derivation in Designware is given in [27].

3.2. Basic Concepts

3.2.1. Specifications A specification is the finite presentation of a theory. The signature of a specification provides the vocabulary for describing objects, operations, and properties in some domain of interest, and the axioms constrain the meaning of the symbols. The theory of the domain is the closure of the axioms under the rules of inference.

Example: Here is a specification for partial orders, using notation adapted from Specware. It introduces a sort E and an infix binary predicate on E , called le , which is constrained by the usual axioms. Although Specware allows higher-order specifications, first-order formulations are sufficient for most purposes.

```
spec Partial-Order is
  sort  $E$ 
  op  $le_$  :  $E, E \rightarrow Boolean$ 
  axiom reflexivity is  $x le x$ 
  axiom transitivity is  $x le y \wedge y le z \implies x le z$ 
  axiom antisymmetry is  $x le y \wedge y le x \implies x = z$ 
end-spec
```

Example: Containers are constructed by a binary join operator and they represent finite collections of elements of some sort E . The specification shown in Figure 1 includes a definition by means of axioms. Operators are required to be total. The constructor clause asserts that the operators $\{empty, singleton, join\}$ construct the sort $Container$, providing the basis for induction on $Container$.

The generic term *expression* will be used to refer to a term, formula, or sentence.

A model of a specification is a structure of sets and total functions that satisfy the axioms. However, for software development purposes we have a less well-defined notion of semantics in mind: each specification denotes a set of possible implementations in

```

spec Container is
  sorts E, Container
  op empty :→ Container
  op singleton : E → Container
  op _join_ : Container, Container → Container
  constructors {empty, singleton, join} construct Container

  axiom ∀(x : Container)(x join empty = x ∧ empty join x = x)
  op _in_ : E, Container → Boolean
  definition of in is
    axiom x in empty = false
    axiom x in singleton(y) = (x = y)
    axiom x in U join V = (x in U ∨ x in V)
  end-definition
end-spec

```

Fig. 1. Specification for Containers

some computational model. Currently we regard these as functional programs. A denotational semantics maps these into classical models.

3.2.2. Morphisms A specification morphism translates the language of one specification into the language of another specification, preserving the property of provability, so that any theorem in the source specification remains a theorem under translation.

A *specification morphism* $m : T \rightarrow T'$ is given by a map from the sort and operator symbols of the *domain* spec T to the symbols of the *codomain* spec T' . To be a specification morphism it is also required that every axiom of T translates to a theorem of T' . It then follows that a specification morphism translates theorems of the domain specification to theorems of the codomain.

Example: A specification morphism from *Partial-Order* to *Integer* is:

```

morphism Partial-Order-to-Integer is
  {E ↪ Integer, le ↪ ≤}

```

Translation of an expression by a morphism is by straightforward application of the symbol map, so, for example, the *Partial-Order* axiom $x \text{ le } x$ translates to $x \leq x$. The three axioms of a partial order remain provable in *Integer* theory after translation.

Morphisms come in a variety of flavors; here we only use two. An *extension* or *import* is an inclusion between specs.

Example: We can build up the theory of partial orders by importing the theory of preorders. The import morphism is $\{E \mapsto E, \text{le} \mapsto \text{le}\}$.

```
spec PreOrder
```

```

sort E
op _le_ : E, E → Boolean
axiom reflexivity is x le x
axiom transitivity is x le y ∧ y le z ⇒ x le z
end-spec

spec Partial-Order
  import PreOrder
  axiom antisymmetry is x le y ∧ y le x ⇒ x = z
end-spec

```

A *definitional extension*, written $A \dashv\rightarrow B$, is an import morphism in which any new symbol in B also has an axiom that defines it. Definitions have implicit axioms for existence and uniqueness. Semantically, a definitional extension has the property that each model of the domain has a unique expansion to a model of the codomain.

Example: *Container* can be formulated as a definitional extension of *Pre-Container*:

```

spec Pre-Container is
  sorts E, Container
  op empty : → Container
  op singleton : E → Container
  op _join_ : Container, Container → Container
  constructors {empty, singleton, join} construct Container
  axiom ∀(x : Container)(x join empty = x ∧ empty join x = x)
end-spec

spec Container is
  imports Pre-Container
  definition of in is
    axiom x in empty = false
    axiom x in singleton(y) = (x = y)
    axiom x in U join V = (x in U ∨ x in V)
  end-definition
end-spec

```

A parameterized specification can be treated syntactically as a morphism.

Example: The specification *Container* can be parameterized on a spec *Triv* with a single sort:

```

spec Triv is
  sort E
end-spec

```

via

parameterized-spec *Parameterized-Container* : $TRIV \rightarrow Container$ is
 $\{E \mapsto E\}$

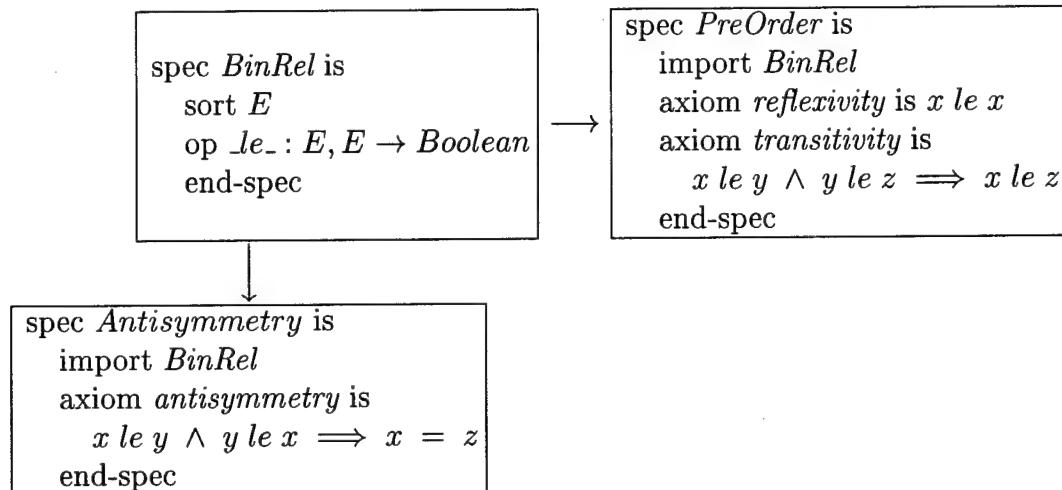
A functorial semantics for first-order parameterized specifications via coherent functors is given in section 4..

3.2.3. The Category of Specs Specification morphisms compose in a straightforward way as the composition of finite maps. It is easily checked that specifications and specification morphisms form a category SPEC. Colimits exist in SPEC and are easily computed. Suppose that we want to compute the colimit of $B \xleftarrow{i} A \xrightarrow{j} C$. First, form the disjoint union of all sort and operator symbols of A , B , and C , then define an equivalence relation on those symbols:

$$s \approx t \text{ iff } (i(s) = t \vee i(t) = s \vee j(s) = t \vee j(t) = s).$$

The signature of the colimit (also known as pushout in this case) is the collection of equivalence classes wrt \approx . The cocone morphisms take each symbol into its equivalence class. The axioms of the colimit are obtained by translating and collecting each axiom of A , B , and C .

Example: Suppose that we want to build up the theory of partial orders by composing simpler theories.



The pushout of *Antisymmetry* \leftarrow *BinRel* \rightarrow *PreOrder* is isomorphic to the specification for *Partial-Order* in Section 2.1. In detail: the morphisms are $\{E \mapsto E, le \mapsto le\}$ from *BinRel* to both *PreOrder* and *Antisymmetry*. The equivalence classes are then $\{\{E, E, E\}, \{le, le, le\}\}$, so the colimit spec has one sort (which

we rename E), and one operator (which we rename le). Furthermore, the axioms of $BinRel$, $Antisymmetry$, and $PreOrder$ are each translated to become the axioms of the colimit. Thus we have *Partial-Order*.

Example: The pushout operation is also used to instantiate the parameter in a parameterized specification [6]. The binding of argument to parameter is represented by a morphism. To form a specification for Containers of integers, we compute the pushout of $Container \leftarrow Triv \rightarrow Integer$, where $Container \leftarrow Triv$ is $\{E \mapsto E\}$, and $Triv \rightarrow Integer$ is $\{E \mapsto Integer\}$.

Example: A specification for sequences can be built up from $Container$, also via pushouts. We can regard $Container$ as parameterized on a binary operator

```
spec BinOp is
  sort E
  op _bop_ : E, E → E
end-spec
```

```
morphism Container-Parameterization : BinOp → Container is
  {E ↪ E, bop ↪ join}
```

and we can define a refinement arrow that extends a binary operator to a semigroup:

```
spec Associativity is
  import BinOp
  axiom Associativity is ((x join y) join z) = (x join (y join z))
end-spec
```

The pushout of $Associativity \leftarrow BinOp \rightarrow Container$, produces a collection specification with an associative join operator, which is *Proto-Seq*, the core of a sequence theory (See Appendix in [27]). By further extending *Proto-Seq* with a commutativity axiom, we obtain *Proto-Bag* theory, the core of a bag (multiset) theory.

3.2.4. Diagrams Roughly, a *diagram* is a graph morphism to a category, usually the category of specifications in our work. For example, the pushout described above started with a diagram comprised of two arrows:

$$\begin{array}{ccc} BinRel & \longrightarrow & PreOrder \\ \downarrow & & \\ Antisymmetry & & \end{array}$$

and computing the pushout of that diagram produces another diagram:

$$\begin{array}{ccc} \text{BinRel} & \longrightarrow & \text{PreOrder} \\ \downarrow & & \downarrow \\ \text{Antisymmetry} & \longrightarrow & \text{Partial-Order} \end{array}$$

A diagram *commutes* if the composition of arrows along two paths with the same start and finish node yields equal arrows.

3.2.5. The Structuring of Specifications Colimits can be used to construct a large specification from a diagram of specs and morphisms. The morphisms express various relationships between specifications, including sharing of structure, inclusion of structure, and parametric structure. Several examples will appear later.

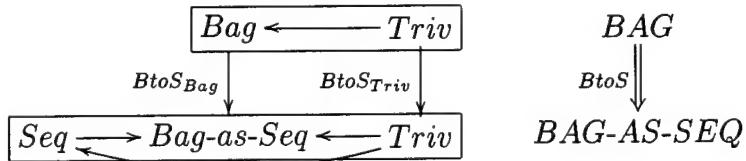
Example: The finest-grain way to compose *Partial-Order* is via the colimit of

$$\begin{array}{ccc} & \text{BinRel} & \\ & \searrow & \downarrow & \swarrow \\ \text{Reflexivity} & & \text{Transitivity} & & \text{Antisymmetry} \end{array}$$

3.2.6. Refinement and Diagrams As described above, specification morphisms can be used to help *structure* a specification, but they can also be used to *refine* a specification. When a morphism is used as a refinement, the intended effect is to reduce the number of possible implementations when passing from the domain spec to the codomain. In this sense, a refinement can be viewed as embodying a particular design decision or property that corresponds to the subset of possible implementations of the domain spec which are also possible implementations of the codomain.

Often in software refinement we want to preserve and extend the structure of a structured specification (versus flattening it out via colimit). When a specification is structured as a diagram, then the corresponding notion of structured refinement is a diagram morphism. A *diagram morphism* M from diagram D to diagram E consists of a set of specification morphisms, one from each node/spec in D to a node in E such that certain squares commute (a functor underlies each diagram and a natural transformation underlies each diagram morphism). We use the notation $D \Rightarrow E$ for diagram morphisms.

Example: A datatype refinement that refines bags to sequences can be presented as the diagram morphism $BtoS : BAG \Rightarrow BAG\text{-AS-SEQ}$:



where the domain and codomain of $BtoS$ are shown in boxes, and the (one) square commutes. Here $Bag\text{-}as\text{-}Seq$ is a definitional extension of Seq that provides an image for Bag theory. Specs for Bag , Seq and $Bag\text{-}as\text{-}Seq$ and details of the refinement can be found in Appendix A of [27]. The interesting content is in spec morphism $BtoS_{Bag}$:

morphism $BtoS_{Bag} : Bag \rightarrow Bag\text{-}as\text{-}Seq$ is

$\{Bag$	\mapsto	$Bag\text{-}as\text{-}Seq,$
$empty\text{-}bag$	\mapsto	$bag\text{-}empty,$
$empty\text{-}bag?$	\mapsto	$bag\text{-}empty?,$
$nonempty?$	\mapsto	$bag\text{-}nonempty?,$
$singleton\text{-}bag$	\mapsto	$bag\text{-}singleton,$
$singleton\text{-}bag?$	\mapsto	$bag\text{-}singleton?,$
$nonsingleton\text{-}bag?$	\mapsto	$bag\text{-}nonsingleton?,$
in	\mapsto	$bag\text{-}in,$
$bag\text{-}union$	\mapsto	$bag\text{-}union,$
$bag\text{-}wfgt$	\mapsto	$bag\text{-}wfgt,$
$size$	\mapsto	$bag\text{-}size\}$

Diagram morphisms compose in a straightforward way based on spec morphism composition. It is easily checked that diagrams and diagram morphisms form a category. Colimits in this category can be computed using left Kan extensions and colimits in SPEC. In the sequel we will generally use the term refinement to mean a diagram morphism.

3.2.7. Logic Morphisms and Code Generation Inter-logic morphisms [24] are used to translate specifications from the specification logic to the logic of a programming language. See [28] for more details. They are also useful for translating between the specification logic and the logic supported by various theorem-provers and analysis tools. They are also useful for translating between the theory libraries of various systems.

3.3. Software Development by Refinement



The development of correct-by-construction code via a formal refinement process is shown to the left. The refinement process starts with a specification S_0 of the requirements on a desired software artifact. Each S_i , $i = 0, 1, \dots, n$ represents a structured specification (diagram) and the arrows \Downarrow are refinements (represented as diagram morphisms). The refinement from S_i to S_{i+1} embodies a design decision which cuts down the number of possible implementations. Finally an inter-logic morphism translates a low-level specification S_n to code in a programming language. Semantically the effect is to narrow down the set of possible implementations of S_n to just one, so specification refinement can be viewed as a constructive process for proving the existence of an implementation of specification S_0 (and proving its consistency).

Clearly, two key issues in supporting software development by refinement are: (1) how to construct specifications, and (2) how to construct refinements. Most of the sequel treats mechanizable techniques for constructing refinements.

3.3.1. Constructing Specifications A specification-based development environment supplies tools for creating new specifications and morphisms, for structuring specs into diagrams, and for composing specifications via importation, parameterization, and colimit. In addition, a software development environment needs to support a large library of reusable specifications, typically including specs for (1) common datatypes, such as integer, sequences, finite sets, etc. and (2) common mathematical structures, such as partial orders, monoids, vector spaces, etc. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific theories, such as resource theories, or generic theories about domains such as satellite control or transportation.

3.3.2. Constructing Refinements A refinement-based development environment supplies tools for creating new refinements. One of our innovations is showing how a library of abstract refinements can be applied to produce refinements for a given specification. In this section, we focus mainly on refinements that embody design knowledge about (1) algorithm design, (2) datatype refinement, and (3) expression optimization. We believe that other types of design knowledge can be similarly expressed

and exploited, including interface design, software architectures, domain-specific requirements capture, and others. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific refinements.

The key concept of this work is the following: abstract design knowledge about datatype refinement, algorithm design, software architectures, program optimization rules, visualization displays, and so on, can be expressed as refinements (i.e. diagram morphisms). The domain of one such refinement represents the abstract structure that is required in a user's specification in order to apply the embodied design knowledge. The refinement itself embodies a design constraint – the effect is a reduction in the set of possible implementations. The codomain of the refinement contains new structures and definitions that are composed with the user's requirement specification.

$$\begin{array}{ccc} A & \Longrightarrow & S_0 \\ \Downarrow & & \Downarrow \\ B & \Longrightarrow & S_1 \end{array}$$

The figure to the left shows the application of a library refinement $A \Rightarrow B$ to a given (structured) specification S_0 . First the library refinement is selected. The applicability of the refinement to S_0 is shown by constructing a *classification arrow* from A to S_0 which classifies S_0 as having A -structure by making explicit how S_0 has at least the structure of A . Finally the refinement is applied by computing the pushout in the category of diagrams. The creative work lies in constructing the classification arrow [25, 26].

3.4. Scaling up

The process of refining specification S_0 described above has three basic steps:

1. select a refinement $A \Rightarrow B$ from a library,
2. construct a classification arrow $A \Rightarrow S_0$, and
3. compute the pushout S_1 of $B \Leftarrow A \Rightarrow S_0$.

The resulting refinement is the cocone arrow $S_0 \Rightarrow S_1$. This basic refinement process is repeated until the relevant sorts and operators of the spec have sufficiently explicit definitions that they can be easily translated to a programming language, and then compiled.

In this section we address the issue of how this basic process can be further developed in order to scale up as the size and complexity of the library of specs and refinements grows. The first key idea is to organize libraries of specs and refinements into *taxonomies*. The second key idea is to support *tactics* at two levels: theory-specific tactics for constructing classification arrows, and task-specific tactics that compose common sequences of the basic refinement process into a larger refinement step.

3.4.1. Design by Classification: Taxonomies of Refinements A productive software development environment will have a large library of reusable refinements, letting the user (or a tactic) select refinements and decide where to apply them. The need arises for a way to organize such a library, to support access, and to support efficient construction of classification arrows. A library of refinements can be organized into *taxonomies* where refinements are indexed on the nodes of the taxonomies, and the nodes include the domains of various refinements in the library. The taxonomic links are refinements, indicating how one refinement applies in a stronger setting than another.

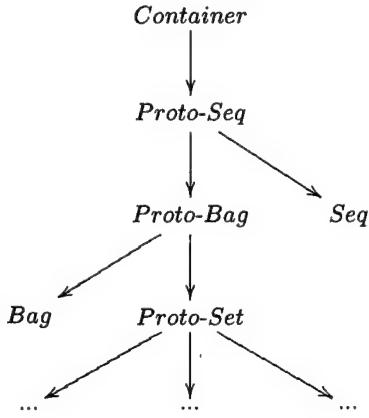


Fig. 2. Taxonomy of Container Datatypes

Figure 2 sketches a taxonomy of abstract datatypes for collections. The arrows between nodes express the refinement relationship; e.g. the morphism from *Proto-Seq* to *Proto-Bag* is an extension with the axiom of commutativity applied to the join constructor of *Proto-Seqs*. Datatype refinements are indexed by the specifications in the taxonomy; e.g. a refinement from (finite) bags to (finite) sequences is indexed at the node specifying (finite) bag theory.

The paper [27] gives a taxonomy of algorithm design theories. The refinements indexed at each node correspond to (families of) program schemes. The algorithm theory associated with a scheme is sufficient to prove the consistency of any instance of the scheme. Nodes that are deeper in a taxonomy correspond to specifications that have more structure than those at shallower levels. Generally, we wish to select refinements that are indexed as deeply in the taxonomy as possible, since the maximal amount of structure in the requirement specification will be exploited. In the algorithm taxonomy, the deeper the node, the more structure that can be exploited in the problem, and the more problem-solving power that can be brought to bear. Roughly

speaking, narrowly scoped but faster algorithms are deeper in the taxonomy, whereas widely applicable general algorithms are at shallower nodes.

Two problems arise in using a library of refinements: (1) selecting an appropriate refinement, and (2) constructing a classification arrow. If we organize a library of refinements into a taxonomy, then the following *ladder construction* process provides incremental access to applicable refinements, and simultaneously, incremental construction of classification arrows.

$$\begin{array}{c} A_0 \xrightarrow{I_0} Spec_0 \\ \Downarrow \quad \Downarrow \\ A_1 \xrightarrow{I_1} Spec_1 \\ \Downarrow \quad \Downarrow \\ A_2 \xrightarrow{I_2} Spec_2 \\ \Downarrow \quad \Downarrow \\ \cdots \quad \cdots \\ \Downarrow \quad \Downarrow \\ A_n \xrightarrow{I_n} Spec_n \end{array}$$

$$\begin{array}{c} A_n \xrightarrow{I_n} Spec_n \\ \Downarrow \quad \Downarrow \\ B_n \xrightarrow{\quad} Spec_{n+1} \end{array}$$

The process of incrementally constructing a refinement is illustrated in the *ladder construction* diagram to the left. The left side of the ladder is a path in a taxonomy starting at the root. The ladder is constructed a rung at a time from the top down. The initial interpretation from A_0 to $Spec_0$ is often simple to construct. The rungs of the ladder are constructed by a constraint solving process that involves user choices, the propagation of consistency constraints, calculation of colimits, and constructive theorem proving [25, 26]. Generally, the rung construction is stronger than a colimit – even though a cocone is being constructed. The intent in contracting $I_i : A_i \Rightarrow Spec_i$ is that $Spec_i$ has sufficient *defined* symbols to serve as the codomain. In other words, the *implicitly* defined symbols in A_i are translated to *explicitly* defined symbols in $Spec_i$.

Once we have constructed a classification arrow $A_n \Rightarrow Spec_n$ and selected a refinement $A_n \Rightarrow B_n$ that is indexed at node A_n in the taxonomy, then constructing a refinement of $Spec_0$ is straightforward: compute the pushout, yielding $Spec_{n+1}$, then compose arrows down the right side of the ladder and the pushout square to obtain $Spec_0 \Rightarrow Spec_{n+1}$ as the final constructed refinement.

Again, rung construction is *not* simply a matter of computing a colimit. For example, there are at least two distinct arrows from *Divide-and-Conquer* to *Sorting*, corresponding to a mergesort and a quicksort – these are distinct cocones and there is no universal sorting algorithm corresponding to the colimit. However, applying the refinement that we select at a node in the taxonomy *is* a simple matter of computing the pushout. For algorithm design the pushout simply instantiates some definition schemes and other axiom schemes.

It is unlikely that a general automated method exists for constructing rungs of the ladder, since it is here that creative decisions can be made. For general-purpose design it seems that users must be involved in guiding the rung construction process. However in domain-specific settings and under certain conditions it will be possible to automate rung construction (as discussed in the next section). Our goal in Designware is to build an interface providing the user with various general automated operations and libraries of standard components. The user applies various operators with the goal of filling out partial morphisms and specifications until the rung is complete. After each user-directed operation, constraint propagation rules are automatically invoked to perform sound extensions to the partial morphisms and specifications in the rung diagram. Constructive theorem-proving provides the basis for several important techniques for constructing classification arrows [25, 26].

3.4.2. Tactics The design process described so far uses primitive operations such as (1) selecting a spec or refinement from a library, (2) computing the pushout/colimit of (a diagram of) diagram morphisms, and (3) unskolemizing and translating a formula along a morphism, (4) witness-finding to derive symbol translations during the construction of classification arrows, and so on. These and other operations can be made accessible through a GUI, but inevitably, users will notice certain patterns of such operations arising, and will wish to have macros or parameterized procedures for them, which we call *tactics*. They provide higher level (semiautomatic) operations for the user.

The need for at least two kinds of tactics can be discerned.

1. *Classification tactics* control operations for constructing classification arrows. The divide-and-conquer theory admits at least two common tactics for constructing a classification arrow. One tactic can be procedurally described as follows: (1) the user selects a operator symbol with a DRO requirement spec, (2) the system analyzes the spec to obtain the translations of the DRO symbols, (3) the user is prompted to supply a standard set of constructors on the input domain D , (4) the tactic performs unskolemization on the composition relation in each Soundness axiom to derive a translations for O_{Ci} , and so on. This tactic was followed in the mergesort derivation.

The other tactic is similar except that the tactic selects constructors for the composition relations on R (versus D) in step (3), and then uses unskolemization to solve for decomposition relations in step (4). This tactic was followed in the quicksort derivation.

A classification tactic for context-dependent simplification provides another example. Procedurally: (1) user selects an expression $expr$ to simplify, (2) type analysis is used to infer translations for the input and output sorts of $expr$, (3) a context

analysis routine is called to obtain contextual properties of *expr* (yielding the translation for *C*), (4) unskolemization and witness-finding are used to derive a translation for *new-expr*.

2. *Refinement tactics* control the application of a collection of refinements; they may compose a common sequence of refinements into a larger refinement step. Planware has a code-generation tactic for automatically applying spec-to-code inter-logic morphisms. Another example is a refinement tactic for context-dependent simplification; procedurally, (1) use the classification tactic to construct the classification arrow, (2) compute the pushout, (3) apply a substitution operation on the spec to replace *expr* with its simplified form and to create an isomorphism. Finite Differencing requires a more complex tactic that applies the tactic for context-dependent simplification repeatedly in order to make incremental the expressions set up by applying the *Expression-and-Function → Abstracted-Op* refinement.

We can also envision the possibility of metatactics that can construct tactics for a given class of tasks. For example, given an algorithm theory, there may be ways to analyze the sorts, ops and axioms to determine various orders in constructing the translations of classification arrows. The two tactics for divide-and-conquer mentioned above are an example.

3.5. Summary

The main message of this section is that a formal software refinement process can be supported by automated tools, and in particular that libraries of design knowledge can be brought to bear in constructing refinements for a given requirement specification. One goal of this section has been to show that diagram morphisms are adequate to capture design knowledge about algorithms, data structures, and expression optimization techniques, as well as the refinement process itself. We showed how to apply a library refinement to a requirement specification by constructing a classification arrow and computing the pushout. We discussed how a library of refinements can be organized into taxonomies and presented techniques for constructing classification arrows incrementally. The examples and most concepts described are working in the Specware, Designware, and Planware systems.

Acknowledgements: The work reported here is the result of extended collaboration with our colleagues at Kestrel Institute. We would particularly like to acknowledge the contributions of Li Mei Gilham, Junbo Liu, Duško Pavlović, and Stephen Westfold.

4. Parametrized specifications

Parametricity is one of the most effective ways to achieve compositionality and reuse in software development. Parametric specifications have been thoroughly analyzed in

the algebraic setting and are by now a standard part of most software development toolkits. However, an effort towards classifying, specifying and refining algorithmic theories, rather than mere datatypes, quickly leads beyond the realm of algebra, and often to full first order theories. We extend the standard semantics of parametric specifications to this more general setting.

The familiar semantic characterization of parametricity in the algebraic case is expressed in terms of the free functor, i.e. using the initial models. In the general case, initial models may not exist, and the free functor is not available. Various syntactic, semantic, and abstract definitions of parametricity have been offered, but their exact relationships are often unclear. Using the methods of categorical model theory, we establish the equivalence of two well known, yet so far unrelated, definitions of parametricity, one syntactic, one semantic. Besides providing support for both underlying views, and a way for aligning the systems based on each of them, the offered general analysis and its formalism open several avenues for future research and applications.

4.1. Introduction

4.1.1. Parametric specifications The idea of *parametric polymorphism* goes back to Strachey [30] and refers to code reusable over any type that may be passed to it as a parameter. If a type is viewed as a set of logical invariants of the data, this idea naturally extends to the software specifications, as the logical theories capturing requirements and allowing their refinement. The idea of parametric specifications was proposed early on and became a standard part of specification theory (cf. e.g. [9, 12, 13] and the references therein).

A standard nontrivial example of a parametric specification is a presentation of the theory of vector spaces, with the theory of fields as its parameter. The idea is that refining the parameter, in this case the subtheory referring to scalars, yields a consistent refinement of the larger theory, usually called the *body*. Formally, we are given a theory VecSp and a distinguished subtheory $\text{Field} \hookrightarrow \text{VecSp}$. The refinement is realized by the pushout in the category of specifications [6, 14].

$$\begin{array}{ccc} \text{Field} & \longrightarrow & \text{VecSp}[\text{Field}] \\ \downarrow & & \downarrow \\ \text{Real} & \longrightarrow & \text{VecSp}[\text{Real}] \end{array}$$

The functoriality of the pushout operation ensures the compositionality of the refinements.

Of course, not every interpretation of one specification in another allows this. For instance, if instead of *Field*, just the theory of rings is taken as the parameter of *VecSp*, some consistent refinements of the parameter will induce inconsistencies in the body. Some models of the parameter therefore do not correspond to models of the body.

Some syntactic parametricity conditions, ensuring that consistent refinements of the parameter induce consistent refinements of the body, were proposed early on [11, 15]. However, the analogous semantic characterizations, ensuring that models of the parameter induce models of the body, were given only in terms of free functors, which only exist for (essentially) algebraic specifications, i.e. those stated using just operations and equations (and simple implications between them). In [11], cofree functors were analyzed as well, but for a general first order theory, they may not exist either. E.g., the theories of fields, Hilbert spaces, or linear orders do not have either initial or final models.

Algebraic specifications do suffice for great many practical tasks and offer a fruitful ground for theory [9]. However, when it comes to systems for code synthesis, like SPECWARE™ [17], where it is essential to compositionally refine and implement not only abstract datatypes, but also abstract algorithmic theories, algebraic specifications become increasingly insufficient, and initial and final semantics do not apply.

On one hand, a syntactic form of parametricity for general specifications has been used in practice and in the literature [12, 13]. On the other hand, in [8], a semantical definition of parametricity was proposed, independent of the existence of initial or final models. However, it seems that neither the semantic characterization of the former nor the syntactic characterization of the latter have been worked out. Abstracting away from the concrete meaning of parametricity, some interesting structures have been built, applicable to parametric specifications in general [7, 29], yet no statement tying together the syntactic and the semantic intuitions seems to have been proved. The purpose of the present paper is to try to bridge this gap, while providing some evidence of the applicability of *categorical model theory* to the study of general software specifications.¹

4.1.2. Elements of categorical model theory The functorial semantics of algebraic theories goes back to the sixties, to Lawvere's thesis [19]. The theory of categorical universal algebra which arose from it is summarized in [23]. An important step beyond algebra is the study of locally presentable categories [10], which come about as the model categories of limit theories, a wider, yet essentially restricted class. The full scope of first order logic was covered by categorical model theory rather slowly,

¹ In contrast, the purported algebraicizing of general specifications in higher order logic by presenting the first order theorems as higher order equations only shifts the problems from the large but familiar area of first order model theory to the scarcely cultivated field of higher order algebra.

throughout the seventies and eighties, as some parts tend to be technically rather demanding. Good accounts of the more accessible parts are [2, 21, 22].

The main idea of functorial semantics is to

- present logical theories as *classifying* categories with structure, so as to
- obtain their models as structure preserving functors to Set , with homomorphisms between them as natural transformations.

The resulting categories of models will always be *accessible*, i.e. have directed colimits and a suitable generating set. Conversely, every accessible category can be obtained as the category of models of a first order theory, possibly infinitary. Categorical model theory is thus the study of accessible categories and the way they arise from theories. There is a very general Stone-type duality between the first order theories, presented as categories, and the induced categories of models [20], but it is quite involved in technical details, and it is not clear whether it can be brought into a practically useful form.

But without going into the formal duality, one can still systematically explore the relationships between the *syntactic* and the *semantic* aspects of theories, by analyzing functors between their categorical presentations. In particular, for any two first order theories \mathbb{A} and \mathbb{B} , presented as classifying categories, one can align the properties of the logical interpretations, which can be captured as functors $F : \mathbb{A} \rightarrow \mathbb{B}$, and the induced forgetful, or “reduct”, functors $F^\# : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$ between the corresponding categories of models.

This is a typical task for semantics of software specifications: analyze how a particular class of syntactical manipulations with theories is reflected on their models, and on the computations that may be built on top. We shall show that a syntactic definition of parametric specification, viewed as a property of the interpretation functor $F : \mathbb{A} \rightarrow \mathbb{B}$, is equivalent to an independent semantic definition, stated in terms of the “reduct” functor $F^\# : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$.

4.1.3. Outline of the paper In the next section, we describe the concrete constructions of classifying categories, explain how interpretations are captured as functors between them, and how the idea of parametricity fits into this setting.

In section 4.3. we list some abstract preliminary results that align the syntactic and semantic properties of functors.

Finally, in section 4.4., we derive the main result: the equivalence of a syntactic form of parametricity, in the spirit of [12, 13], and a semantic form, as in [8], both adapted only to a common categorical setting.

4.2. Theories and models, categorically

4.2.1. Classifying categories The simplest classifying category is the *Lawvere clone* $\mathbb{C}_{\mathcal{T}}$ of an algebraic theory \mathcal{T} , say single sorted. Its objects can be viewed as natural numbers (*viz* the arities), while a morphism from m to n is an n -tuple of the elements of the free algebra in m generators, i.e. a function $n \rightarrow Tm$, where T denotes the free algebra constructor.² A crucial observation from Lawvere’s thesis [19] is that $\mathbb{C}_{\mathcal{T}}$ classifies \mathcal{T} -algebras, in the sense that they exactly correspond to the product preserving functors $\mathbb{C}_{\mathcal{T}} \rightarrow \text{Set}$, while the \mathcal{T} -homomorphisms correspond to the natural transformations between them. Indeed, since n in $\mathbb{C}_{\mathcal{T}}$ appears as the product of n copies of 1, the product preservation ensures that the functors $\mathbb{C}_{\mathcal{T}} \rightarrow \text{Set}$ trace the operations with the correct arities. The equations of \mathcal{T} are then enforced by the functoriality. Detailed explanations of the functorial semantics of algebraic theories can be found e.g. in [23].

If models of more general theories are to be captured as functors, some additional preservation properties will be needed, in order to enforce satisfaction of not necessarily equational formulas, that may express more than mere commutativity conditions. There are several well known frameworks for building suitable classifying categories and developing functorial semantics for general first order theories, the most “categorical” being probably sketches [4, 21]. We shall however work in the setting of *coherent categories* [22], closest to the original geometric spirit of categorical logic, because they seem to allow the quickest and perhaps the most intuitive approach to the matters presently of interest.

4.2.2. Coherent categories Let \mathcal{T} be a multisorted first order theory with equality. For simplicity, we assume that it is purely relational: operations are captured by their graphs. Moreover, \mathcal{T} is assumed to be generated by a set of axioms in *coherent* logic, i.e. using finitary \wedge and \vee , including the empty ones, \top and \perp , and the quantifier \exists . The underlying logic can be classical or intuitionistic. We cannot go into the details here, but reducing finitary first order logic to its coherent fragment is a fairly standard technical device (see [2, 1, 22] and especially the informative introduction of [21]). The extension to infinitary logic is justified by stable and natural categories of models and is routinely handled by extending the classifying constructions. However, some of the proofs presented below essentially depend on the finiteness assumption.

Formally, the theory \mathcal{T} can be viewed as a preorder: the underlying set $|\mathcal{T}|$ of well-formed formulas is generated by its language, while the entailment preorder \vdash is generated by its axioms. The rough idea is to capture the well-formed formulas of \mathcal{T}

² So if \mathcal{T} is presented by the monad T , the classifier $\mathbb{C}_{\mathcal{T}}$ is the dual of the induced Kleisli category, restricted to natural numbers.

as the objects of the classifying category $\mathbb{C}_{\mathcal{T}}$, and the theorems of \mathcal{T} as the morphisms of $\mathbb{C}_{\mathcal{T}}$.

The passage from the formulas of \mathcal{T} to the objects of $\mathbb{C}_{\mathcal{T}}$ requires an adjustment: the formulas must be viewed modulo variable renaming, i.e. α -conversion $\phi(x) \sim \phi(y)$, where x and y are vectors of variables. Note that this is not a congruence with respect to the logical operations, because e.g. $\phi(x) \wedge \phi(y) \not\sim \phi(x) \wedge \phi(x)$.

The passage from theorems of \mathcal{T} to morphisms of $\mathbb{C}_{\mathcal{T}}$ requires a similar adjustment: modulo the logical equivalence $\varphi \dashv\vdash \psi$, which means that $\varphi \vdash \psi$ and $\psi \vdash \varphi$. The definition is thus

$$\begin{aligned} |\mathbb{C}_{\mathcal{T}}| &= |\mathcal{T}| / \sim \\ \mathbb{C}_{\mathcal{T}}(\alpha(x), \beta(y)) &= \{ \vartheta(x, y) \in \mathcal{T} \mid \vartheta(x, y) \vdash \alpha(x) \wedge \beta(y), \\ &\quad \alpha(x) \vdash \exists y. \vartheta(x, y), \\ &\quad \vartheta(x, y') \wedge \vartheta(x, y'') \vdash y' \equiv y'' \} / \dashv\vdash \end{aligned}$$

where x and y are disjoint strings of variables, always available by renaming³, and \equiv is the equality predicate in \mathcal{T} . The identities in $\mathbb{C}_{\mathcal{T}}$ are induced by the equality predicates, and the composition of $\vartheta(x, y)$ and $\varrho(y, z)$ is $\exists y. \vartheta(x, y) \wedge \varrho(y, z)$.

The logical structure of \mathcal{T} induces the categorical structure of $\mathbb{C}_{\mathcal{T}}$:

- *finite limits* are constructed using the conjunction and the variable tupling, starting from the true predicates $\top(x)$ over each sort;
- *regular epi-mono factorisations* are constructed using the existential quantifier; and finally
- *joins of the subobjects* correspond to the disjunctions.

These three structural components constitute a coherent category and are preserved by coherent functors. Theories in coherent logic generate coherent classifying categories; conversely, each small coherent category classifies a coherent theory. Coherent functors preserve the truth of the theorems in coherent logic. The reader may wish to work out the details of this correspondence or to consult some of the mentioned references.

A reader familiar with the functorial semantics of algebra has perhaps already noticed that the coherent classifier of an algebraic theory contains the corresponding Lawvere clone as a full subcategory, namely the one spanned by the true formulas $\top(x)$, one for each arity x . Indeed, the coherent classifier of an algebraic theory is the coherent completion of its Lawvere clone. The coherent classifiers have a richer set of objects, in order to impose the preservation of more general axioms; but simpler theories can be captured by smaller classifiers.

³ By abuse of notation, $\alpha(x)$, $\beta(y)$ and $\vartheta(x, y)$ denote their equivalence classes $[\alpha]$, $[\beta]$ and $[\vartheta]$ modulo \sim .

4.2.3. Interpretations and models The upshot of coherent classifying categories is thus that the coherent functors, preserving the coherent structure, preserve the coherent logic, and thus enforce the satisfaction of the coherent theorems, represented as the morphisms in coherent categories. A coherent functor $\mathbb{C}_{\mathcal{T}} \rightarrow \mathbb{C}_{\mathcal{U}}$ can thus be viewed as a sound interpretation of the theory \mathcal{T} in the theory \mathcal{U} . But since every small coherent category \mathbb{A} can be obtained as the classifier $\mathbb{C}_{\mathcal{T}}$ of some coherent theory \mathcal{T} , every coherent functor $F : \mathbb{A} \rightarrow \mathbb{B}$ can be understood logically, as such an interpretation.

Although it is not small, \mathbf{Set} has all the coherent structure, and the coherent functors $\mathbb{C}_{\mathcal{T}} \rightarrow \mathbf{Set}$ are exactly the models of \mathcal{T} . The natural transformations are the \mathcal{T} -homomorphisms, preserving all the definable operations. For every small coherent \mathbb{A} , we shall denote by $\mathbf{Mod}(\mathbb{A})$ the category of coherent functors $\mathbb{A} \rightarrow \mathbf{Set}$. This is the category of models of \mathbb{A} . As pointed out before, categories of the form $\mathbf{Mod}(\mathbb{A})$ are accessible, and by allowing infinite disjunctions, one could get (an equivalent version of) every accessible category in this form [2, ch. 5].

On the other hand, by precomposition, every coherent functor $F : \mathbb{A} \rightarrow \mathbb{B}$ induces a “*reduct*” $F^{\#} : \mathbf{Mod}(\mathbb{B}) \rightarrow \mathbf{Mod}(\mathbb{A})$, reinterpreting a model $N : \mathbb{B} \rightarrow \mathbf{Set}$ of \mathbb{B} as a model $NF : \mathbb{A} \rightarrow \mathbf{Set}$ of \mathbb{A} . This is the arrow part of the \mathbf{Mod} -construction, which yields an indexed category $\mathbf{Mod} : \mathbf{Coh}^{op} \rightarrow \mathbf{CAT}$, where \mathbf{Coh} is the category of small coherent categories and functors, and \mathbf{CAT} is the metacategory of categories. \mathbf{Mod} thus assigns a semantics to each coherent theory \mathcal{T} , classified by a coherent category $\mathbb{C}_{\mathcal{T}}$; in other words, it maps each theory \mathcal{T} to its category of models, captured as coherent functors $\mathbb{C}_{\mathcal{T}} \rightarrow \mathbf{Set}$.

The semantical functor \mathbf{Mod} is an instance of a *specification frame* in the sense of Ehrig and Große-Rhode [8]. Specification frames are indexed categories, construed as some abstract model category assignments, like \mathbf{Mod} . In these terms, Ehrig and Große-Rhode proposed a semantical definition of parametric specifications, which will be analyzed in the sequel.

4.2.4. Parametrized specifications as functors:

syntactic vs semantic definitions A reader unfamiliar with coherent logic may wish to write down, as a quick exercise, say, the coherent theories of fields and vector spaces and analyze their classifying categories. The classifying category \mathbf{Field} is of course a subcategory of the classifying category \mathbf{VecSp} . The obvious functor $\mathbf{Field} \hookrightarrow \mathbf{VecSp}$ is full and faithful. This means that the theory of vector spaces is conservative over the theory of fields: no new theorems about the scalars can be proved using the vectors. Moreover, $\mathbf{Field} \hookrightarrow \mathbf{VecSp}$ is also a powerful functor: each subobject of an object in the image is also in the image. This means that every predicate on scalars, expressible in the theory of vector spaces, is already expressible in the theory of fields.

The embedding $\text{Field} \hookrightarrow \text{VecSp}$ is a typical parametric specification, defined syntactically, as in [12, 13]. Viewed in the setting of classifying categories, a parametric specification is thus a coherent functor $F : \mathbb{A} \rightarrow \mathbb{B}$, which is full, faithful and powerful.

On the semantic side, as already mentioned, Ehrig and Große-Rhode [8] have proposed an abstract definition of parametricity, applicable to the functor $\text{Mod} : \text{Coh}^{\text{op}} \rightarrow \text{CAT}$. Omitting the presentation details, a parametric specification is, according to this definition, an interpretation $F : \mathbb{A} \rightarrow \mathbb{B}$, such that the induced functor $F^{\#} : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$ is a retraction, i.e. there is a functor $\Phi : \text{Mod}(\mathbb{A}) \rightarrow \text{Mod}(\mathbb{B})$ with $F^{\#} \circ \Phi \cong \text{Id}$. In words, Φ maps each model M of the parameter \mathbb{A} into a model $N = \Phi M$ of the body \mathbb{B} in such a way that the forgetful functor $F^{\#}$ restores an isomorphic copy⁴ of M . Such a functor Φ , which nondestructively expands a model, is said to be *persistent* [9, sec. 10B].

In the present paper, we shall show that the above two definitions are roughly equivalent: a coherent functor $F : \mathbb{A} \rightarrow \mathbb{B}$ is full, faithful and powerful if and only if $F^{\#} : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$ is a retraction, in the strong sense that every splitting of its object part can be refined, by taking quotients, into a splitting functor.

Completeness view.

When an indexed family of sets $\{B_x | x \in A\}$ is represented as a function $f : B \rightarrow A$, with $B_x = f^{-1}(x)$, an indexed element $b \in \prod_{x \in A} B_x$ becomes a splitting $\phi : A \rightarrow B$, $f \circ \phi = \text{id}$, with $b_x = \phi(x) \in B_x$.

Similarly, a specification \mathbb{B} parametrized over \mathbb{A} can be thought of as a family of the instances of \mathbb{B} indexed over the instances of \mathbb{A} . In particular, the functor $F^{\#} : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$ can be construed as a family of \mathbb{B} -models indexed over \mathbb{A} -models. A splitting $\Phi : \text{Mod}(\mathbb{A}) \rightarrow \text{Mod}(\mathbb{B})$, $F^{\#} \circ \Phi \cong \text{Id}$, then becomes an indexed model of \mathbb{B} , parametrized over \mathbb{A} .

According to this view, a persistent functor is thus an indexed model. The parametricity of theories lifts to the parametricity of their models: the semantical definition of parametric specification, described above, boils down to the requirement that there is a parametric model of the body indexed over the models of the parameter.

The equivalence of the semantic and the syntactic definitions of parametricity, which we are about to establish, thus becomes a soundness-and-completeness theorem, in indexed form.

⁴ The original definition actually requires that M is recovered on the nose, i.e. that the strict equality $F^{\#} \circ \Phi = \text{Id}$ holds. But in abstract functorial calculus, this is almost never possible.

4.3. Syntactic vs semantic properties of functors

4.3.1. Preliminaries We begin by listing some useful terminology and facts from the general functorial calculus.

Definition 1. A functor $F : \mathbb{A} \rightarrow \mathbb{B}$ is said to be

embedding: if it is full and faithful;

subcovering: if for every object $B \in \mathbb{B}$ there is a finite diagram D in \mathbb{B} , such that

- (1) B is the colimit of D , and (2) for every node D_i of D there is some A_i in \mathbb{A} and a monic $D_i \rightarrowtail FA_i$ in \mathbb{B} ;

subobject covering: if every $B \in \mathbb{B}$ is a subobject of some FA , $A \in \mathbb{A}$ (in other words, if it is subcovering and the diagrams D can be chosen to have one node and no edges);

powerful: if all subobjects⁵ of FA in \mathbb{B} lie in the image of F . More precisely, for every monic $D \xrightarrow{d} FA$ in \mathbb{B} there is a monic $S \xrightarrow{s} A$ in \mathbb{A} and an isomorphism $i : D \xrightarrow{\sim} FS$ such that $d = Fs \circ i$;

retraction: if it has a right inverse (i.e., there is $G : \mathbb{B} \rightarrow \mathbb{A}$ with $FGM \cong M$ for all $M \in \mathbb{A}$);

uniform retraction: if it is a retraction, and every splitting of its arrow part refines to a right inverse (more precisely, if $\Gamma : |\mathbb{A}| \rightarrow |\mathbb{B}|$ is such that $F\Gamma M \cong M$, $M \in \mathbb{A}$ then there is a functor $G : \mathbb{A} \rightarrow \mathbb{B}$, where GM is a quotient of ΓM and $FGM \cong M$);

(co)reflection: if it has a right inverse right (resp. left) adjoint.

Lemma 1. A powerful and subobject covering functor is essentially surjective.

Lemma 2. $F : \mathbb{A} \rightarrow \mathbb{B}$ is faithful if and only if

$$F(\varphi) \vdash F(\psi) \implies \varphi \vdash \psi \tag{1}$$

As the converse of (1) is always true, a faithful coherent functor F always induces an “order isomorphism” on the subobject lattices.

To prove lemma 2, use the fact that $\varphi \vdash \psi$ if and only if $\varphi = \varphi \wedge \psi$.

Proposition 1. A coherent functor must be full as soon as it is both faithful and powerful.

⁵ Recall that subobjects are isomorphism classes of monics.

Proof. Since F is powerful, the graph of any $h : FA \rightarrow FA'$ must be in the essential image of F : there must be a monic $\kappa \rightarrowtail A \times B$ in \mathbb{A} the F -image of which is isomorphic with the graph $\chi = \langle \text{id}, h \rangle : FA \rightarrow FA \times FB$. The relation $F\kappa$ thus satisfies;

$$\begin{aligned} \delta_{FA} &\vdash F\kappa ; F\kappa^{op} \\ F\kappa^{op} ; F\kappa &\vdash \delta_{FB} \end{aligned}$$

which respectively tell that it is total and single valued. Taking into account that for the identity relation $\delta = \langle \text{id}, \text{id} \rangle$ holds $\delta_{FX} = F\delta_X$, and using (1), we conclude that κ is a total and single valued relation in \mathbb{A} . In any regular category, such a relation must be isomorphic to one in the form $\langle \text{id}, k \rangle : A \rightarrow A \times B$. Since clearly $F\langle \text{id}, k \rangle = \langle \text{id}, h \rangle$, we conclude that $Fk = h$. \square

4.3.2. Basic results In the sequel, we assume that $F : \mathbb{A} \rightarrow \mathbb{B}$ is a coherent functor between coherent categories, and $F^{\#} : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$ is the functor induced by the precomposition. We use and extend some results from [22]. Note that some of them essentially depend on strong model theoretic assumptions, such as compactness. The proofs are thus largely non-constructive, as they depend on the axiom of choice.

Proposition 2. F is faithful if and only if $F^{\#}$ is essentially surjective.

Proof. By lemma 2, F is faithful if and only if

$$F\varphi \vdash F\psi \iff \varphi \vdash \psi$$

By the completeness theorem [22, thm. 5.1.7] $F\varphi \vdash F\psi$ holds if and only if

$$\forall N \in \text{Mod}(\mathbb{B}). NF\varphi \subseteq NF\psi$$

whereas $\varphi \vdash \psi$ holds if and only if

$$\forall M \in \text{Mod}(\mathbb{A}). M\varphi \subseteq M\psi$$

The last two statements are clearly equivalent if $F^{\#}$ is essentially surjective, i.e.

$$\forall M \in \text{Mod}(\mathbb{A}) \exists N \in \text{Mod}(\mathbb{B}). M \cong F^{\#}N$$

Conversely, if there is $M \in \text{Mod}(\mathbb{A})$ different from $F^{\#}N$ for all $N \in \text{Mod}(\mathbb{B})$, one can use compactness to construct a formula ψ such that $NF\psi$ is true for all models N of \mathbb{B} , whereas $M\psi$ is not. \square

Definition 2. $F^\# : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$ is said to be subfull if every \mathbb{A} -homomorphism $h : F^\# N' \rightarrow F^\# N''$ preserves all \mathbb{B} -subobjects, i.e. for every monic $D \xrightarrow{d} FA$ in \mathbb{B} holds

$$hA(N'D) \subseteq N''D$$

$$\begin{array}{ccc} N'D & \dashrightarrow & N''D \\ N'd \downarrow & & \downarrow N''d \\ N'FA & \xrightarrow{hA} & N''FA \end{array} \quad (2)$$

Proposition 3. F is powerful if and only if $F^\#$ is subfull.

Proof. By definition, F is powerful if and only if every $D \xrightarrow{d} FA$ is in the essential image of F , i.e. $d \cong Fs$ for some $S \xrightarrow{s} A$. So (2) must commute because it is isomorphic with the square

$$\begin{array}{ccc} N'FS & \xrightarrow{hs} & N''FS \\ N'Fs \downarrow & & \downarrow N''Fs \\ N'FA & \xrightarrow{hA} & N''FA \end{array}$$

which commutes by the naturality of h .

The other way around, the fact that the subfullness of $F^\#$, i.e. the commutativity of squares (2) implies that F is powerful is one of the main constituents of the Makkai-Reyes conceptual completeness theorem [22, ch. 7§1]. The proof can be extracted from [22, thms. 7.1.4–4'], and essentially depends on compactness. \square

Proposition 4. F is subcovering if and only if $F^\#$ is faithful.

Proof. Suppose F is subcovering and let $F^\# g = F^\# h$ for some \mathbb{B} -homomorphisms $g, h : N' \rightarrow N''$. The equation $F^\# g = F^\# h$ means that $gFA = hFA : N'FA \rightarrow N''FA$ for all $A \in \mathbb{A}$.

I claim that then $gB = hB : N'B \rightarrow N''B$ must hold for every $B \in \mathbb{B}$. Since F is subcovering, for each B there is a finite diagram D , with (1) a colimit cocone to B ,

i.e. a jointly epimorphic family $\{D_i \xrightarrow{b_i} B\}_{i=1}^n$, and (2) the inclusions $\{D_i \xrightarrow{d_i} FA_i\}_{i=1}^n$ for some objects $A_1, \dots, A_n \in \mathbb{A}$. Hence

$$\begin{array}{ccc}
 N'FA_i & \xrightarrow{\substack{gFA_i = \\ hFA_i}} & N''FA_i \\
 \uparrow N'd_i & & \uparrow N''d_i \\
 N'D_i & \xrightarrow{\substack{gD_i \\ hD_i}} & N''D_i \\
 \downarrow N'b_i & & \downarrow N''b_i \\
 N'B & \xrightarrow{\substack{gB \\ hB}} & N''B
 \end{array} \tag{3}$$

Naturality of g and h now yields

$$\begin{aligned}
 N''d_i \circ gD_i &= gFA_i \circ N'd_i \\
 &= hFA_i \circ N'd_i \\
 &= N''d_i \circ hD_i
 \end{aligned}$$

But since models are left exact, each $N''d_i$ is still a monic, and therefore $gD_i = hD_i$, for all $i = 1, \dots, n$.

Using naturality again, we get

$$\begin{aligned}
 gB \circ N'b_i &= N''b_i \circ gD_i \\
 &= N''b_i \circ hD_i \\
 &= hB \circ N'b_i
 \end{aligned}$$

But since models preserve the finite unions of subobjects $\{N'b_i\}_{i=1}^n$ must be jointly monic again, and therefore $g_B = h_B$. Thus $g = h$, and $F^\#$ is faithful.

For the converse, one assumes that there is $B \in \mathbb{B}$ not subcovered by F , and, using compactness, constructs models N' and N'' and two homomorphisms $g \neq h : N' \rightarrow N''$ such that $F^\#g = F^\#h$. The details are in [22, thms. 7.1.6–6']. \square

Logical meaning.

Proposition 2 tells that each \mathbb{A} -model extends back along $F^\#$ to some \mathbb{B} -model if and only if $F : \mathbb{A} \rightarrow \mathbb{B}$ is faithful. However, this does not guarantee that every \mathbb{A} -homomorphism between \mathbb{A} -models will extend to a \mathbb{B} -homomorphism between their

extensions. Indeed, according to proposition 3, a necessary condition for this is that $F : \mathbb{A} \rightarrow \mathbb{B}$ is powerful.

Together, these conditions provide a basis for aligning syntactical and the semantical definitions of parametricity, as described in section 4.2.4..

4.4. Characterizing parametric specifications

Theorem 1. *For a coherent functor $F : \mathbb{A} \rightarrow \mathbb{B}$ and the induced “reduct” $F^\# : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$, the following statements are equivalent.*

- (a) *F is a powerful embedding.*
- (b) *$F^\#$ is subfull and essentially surjective.*
- (c) *$F^\#$ is a uniform retraction.*

If $\text{Mod}(\mathbb{B})$ has coproducts, then the above conditions are also equivalent with

- (d) *$F^\#$ is coreflection.*

Note that, since $\text{Mod}(\mathbb{B})$ is finitely accessible, it has coproducts if and only if it is locally finitely presentable, i.e. when \mathbb{B} classifies an essentially algebraic theory [2, sec. 3D].

Proof. (a) \Leftrightarrow (b) By proposition 1, it suffices to check that F is faithful and powerful. By proposition 2, F is faithful if and only if $F^\#$ is essentially surjective. By proposition 3, F is powerful if and only if $F^\#$ is subfull.

To simplify the proof of (b) \Rightarrow (c), we shall freely use the established equivalence (a) \Leftrightarrow (b). Given that $F^\#$ is essentially surjective and subfull, we thus know that F is full, faithful and powerful. Using all that, we define $\Phi : \text{Mod}(\mathbb{A}) \rightarrow \text{Mod}(\mathbb{B})$, such that $F^\# \circ \Phi \cong \text{Id}$.

Since $F^\#$ is essentially surjective, for every M in $\text{Mod}(\mathbb{A})$, there is some L in $\text{Mod}(\mathbb{B})$ such that $M \cong F^\# L$. But the homomorphisms to or from M may not extend to every such L , so we cannot simply take $\Phi M = L$.

On the other hand, like any functor, $M : \mathbb{A} \rightarrow \text{Set}$ has the right Kan extension, a functor $F_\# M : \mathbb{B} \rightarrow \text{Set}$ [18], defined

$$F_\# M(B) = \lim_{\leftarrow} M \circ \text{Cod}(B/F) \quad (4)$$

where B/F is the comma category, spanned by the arrows in the form $B \xrightarrow{a} FA$ in \mathbb{B} . A morphism from $B \xrightarrow{a} FA$ to $B \xrightarrow{c} FC$ is an arrow $g : A \rightarrow C$ in \mathbb{A} such that $Fg \circ a = c$. The image of $B \in \mathbb{B}$ along $F_\# M$ is thus the limit of the diagram $B/F \xrightarrow{\text{Cod}} \mathbb{A} \xrightarrow{M} \text{Set}$.

The construction $F_\#$ is functorial and it is not hard to see that $F^\# \circ F_\# \cong \text{Id}$ holds if and only if F is faithful. So $F_\# M$ might be a candidate for ΦM . But the assumption that $M : \mathbf{A} \rightarrow \mathbf{Set}$ is coherent does not generally follow for $F_\# M : \mathbf{B} \rightarrow \mathbf{Set}$. The $F_\#$ -image of an \mathbf{A} -model M may not be a \mathbf{B} -model, and the functor $F_\# : \mathbf{Set}^{\mathbf{A}} \rightarrow \mathbf{Set}^{\mathbf{B}}$ does not restrict to a functor $\mathbf{Mod}(\mathbf{A}) \rightarrow \mathbf{Mod}(\mathbf{B})$.

But the desired model $\Phi M : \mathbf{B} \rightarrow \mathbf{Set}$ can actually be “interpolated” between the Kan extension $F_\# M : \mathbf{B} \rightarrow \mathbf{Set}$, and the arbitrary model $L : \mathbf{B} \rightarrow \mathbf{Set}$ such that $F^\# L \cong M$.

First of all, since $F^\# \dashv F_\#$, every $F^\# L \rightarrow M$ induces $L \rightarrow F_\# M$. Given, as above $M \cong F^\# L$, for every $a : B \rightarrow FA$ in \mathbf{B} , there is $La : LB \rightarrow LFA \cong MA$. Hence a cone $\langle La \rangle_{a \in B/F} : LB \rightarrow M \circ \text{Cod}(B/F)$. By definition (4), this cone induces a unique arrow $\phi B : LB \rightarrow F_\# M(B)$.

Let the functor $\Phi M : \mathbf{B} \rightarrow \mathbf{Set}$ be defined as the monic image of $\phi : L \rightarrow F_\# M$, i.e.

$$\phi B : LB \longrightarrow \Phi M(B) \hookrightarrow F_\# M(B) \quad (5)$$

This ΦM will indeed be a model. Although $F_\# M : \mathbf{B} \rightarrow \mathbf{Set}$ is not a model, when $F : \mathbf{A} \rightarrow \mathbf{B}$ and $M : \mathbf{A} \rightarrow \mathbf{Set}$ preserve (finite) limits, then $F_\# M : \mathbf{B} \rightarrow \mathbf{Set}$ weakly preserves them: for every (finite) diagram $\Delta : I \rightarrow \mathbf{B}$, the set $F_\# M(\lim \Delta)$ is a weak limit of $F_\# M(\Delta)$ and thus contains $\lim F_\# M(\Delta)$ as a retract.

Together with the coherence of $L : \mathbf{B} \rightarrow \mathbf{Set}$, this weak preservation property of $F_\# M$ suffices for the coherence of $\Phi M : \mathbf{B} \rightarrow \mathbf{Set}$. E.g., it preserves the products because the map from $\Phi M(B) \times \Phi M(D)$ to $\Phi M(B \times D)$ on

$$\begin{array}{ccccc} LB \times LD & \longrightarrow & \Phi M(B) \times \Phi M(D) & \hookrightarrow & F_\# M(B) \times F_\# M(D) \\ \downarrow \cong & & \downarrow & & \downarrow \\ L(B \times D) & \longrightarrow & \Phi M(B \times D) & \hookrightarrow & F_\# M(B \times D) \end{array} \quad (6)$$

must be both surjective and injective.

The object part of $\Phi : \mathbf{Mod}(\mathbf{A}) \rightarrow \mathbf{Mod}(\mathbf{B})$ is thus determined by (5). Notice that Φ is not unique, as the definition depends on the choice of L , $F^\# L \cong M$.

To define the arrow part of Φ , take an arbitrary \mathbf{A} -homomorphism $h : M' \rightarrow M''$. It surely induces a natural transformation $F_\# h : F_\# M' \rightarrow F_\# M''$, and we can find \mathbf{B} -models L' and L'' that map by $F^\#$ to M' and M'' , and determine \mathbf{B} -models $\Phi M'$ and $\Phi M''$; but $h : M' \rightarrow M''$ in general does not lift to a homomorphism $L' \rightarrow L''$. However, $\Phi h : \Phi M' \rightarrow \Phi M''$ can be derived from $F_\# h : F_\# M' \rightarrow F_\# M''$ alone.

To simplify notation, write $N' = \Phi M'$ and $N'' = \Phi M''$ and $k = \Phi h$ for the desired homomorphism.

We are given a natural family $hA : M'A \rightarrow M''A$ and we want to extend it to $kB : N'B \rightarrow N''B$, so that $kFA = hA$. In other words, we have the subfamily of functions $kFA : N'FA \rightarrow N''FA$, $A \in \mathbb{A}$, and we need to complete it to a natural family $kB : N'B \rightarrow N''B$, $B \in \mathbb{B}$.

Consider first, for each $B \in \mathbb{B}$, the set \mathcal{E}_B of regular epimorphisms $e : B \rightarrow FA_e$ in \mathbb{B} . The e -th component of the limit cone $F_\#M(B) \rightarrow M \circ \text{Cod}(B/F)$ is a function $F_\#M(B) \rightarrow MA_e$. Hence the map

$$F_\#M(B) \rightarrow \prod_{e \in \mathcal{E}_B} MA_e \quad (7)$$

Since $F : \mathbb{A} \rightarrow \mathbb{B}$ is powerful, this map is injective. By postcomposing (5) with it, one gets

$$\langle Le \rangle_{e \in \mathcal{E}_B} : LB \rightarrow \Phi M(B) \hookrightarrow \prod_{e \in \mathcal{E}_B} LFA_e \quad (8)$$

because $MA_e = LFA_e$. Of course, since L is coherent, each $Le : LB \rightarrow LFA_e$ is a surjection. The set $\Phi M(B)$ can thus also be obtained by taking the product of all sets LFA_e , such that there is some regular epi $e : B \rightarrow FA_e$ in \mathbb{B} , and then extracting from this product the image of the tuple formed by all epis $Le : LB \rightarrow LFA_e$.

The construction of $kB : N'B \rightarrow N''B$ now proceeds by the following steps:

(i) define a function

$$\kappa B : N'B \rightarrow \wp(N''B)$$

such that

$$\kappa FA(x) = \{hA(x)\}$$

- (ii) show that $\kappa B(x)$ is nonempty for every $x \in N'B$;
- (iii) show that $\kappa B(x)$ has at most one element for every $x \in N'B$; writing $kB(x)$ for the only element of $\kappa B(x)$, we get the function $kB : N'B \rightarrow N''B$;
- (iv) prove that the obtained family $kB : N'B \rightarrow N''B$, $B \in \mathbb{B}$ is natural, i.e. forms $k : N' \rightarrow N''$.

(i) Using the same set \mathcal{E}_B of regular epis $e : B \rightarrow FA$ as above, define

$$\begin{aligned} \kappa^e B(x) &= (N''e)^{-1} \circ hA \circ N'e(x) \\ \kappa B(x) &= \bigcap_{e \in \mathcal{E}_B} \kappa^e B(x) \end{aligned}$$

For $B = FA$, $\kappa^{\text{id}} FA(x) = \{hA(x)\}$. Moreover, for every $e \in \mathcal{E}_{FA}$ holds

$$\kappa^{\text{id}} FA(x) \subseteq \kappa^e FA(x) \quad (9)$$

Indeed, since F is full, the naturality of h implies that the square

$$\begin{array}{ccc} N'FA & \xrightarrow{hA} & N''FA \\ N'e \downarrow & & \downarrow N''e \\ N'F\tilde{A} & \xrightarrow{h\tilde{A}} & N''F\tilde{A} \end{array}$$

commutes. Hence (9), and thus $\kappa FA(x) = \{hA(x)\}$, as asserted.

(ii) For every $B \in \mathbb{B}$, the set \mathcal{E}_B is nonempty because it surely contains the regular epi part $B \twoheadrightarrow FI \rightarrowtail F1 \cong 1$. $F1$ is terminal because F is coherent; the regular image of $B \rightarrow F1$ is in the image of F because F is powerful.

Moreover, since N'' is coherent, and $B \twoheadrightarrow FI$ is a cover (regular epi) $N''B \rightarrow N''FI$ must be a surjection. So if $N''B$ is empty, $N''FI$ must be empty, and hence $N'FI$ must be empty, because there is a function $hI : N'FI \rightarrow N''FI$. But there is also a function $N'B \rightarrow N'FI$, and thus $N'B$ must be empty as well, so there is a unique $kB : N'B \rightarrow N''B$, and we are done.

With no loss of generality, we can thus assume that $N''B$ is nonempty. Since $N''e : N''B \rightarrow NFA$, $e \in \mathcal{E}_B$, is a surjection, all NFA are nonempty, and moreover, every $\kappa^e(x) = (N''e)^{-1} \circ hA \circ N'e(x)$ is nonempty.

Finally, for any $e_0 : B \twoheadrightarrow FA_0$ and $e_1 : B \twoheadrightarrow FA_1$ from \mathcal{E}_B the intersection $\kappa^{e_0} \cap \kappa^{e_1}$ is nonempty as well. Toward a proof, consider the pair $\langle e_0, e_1 \rangle : B \rightarrow FA_0 \times FA_1 \cong F(A_0 \times A_1)$ in \mathbb{B} . Factoring, and using once again the assumption that F is powerful, we get $e_2 : B \twoheadrightarrow FA_2$, with a pair $\langle p_0, p_1 \rangle : A_2 \rightarrow A_0 \times A_1$ in \mathbb{A} such that $e_i = Fp_i \circ e_2$, $i = 0, 1$. But $N''e_i = N''Fp_i \circ N''e_2$ implies

$$\kappa^{e_2}(x) \subseteq \kappa^{e_0}(x) \cap \kappa^{e_1}(x)$$

for all $x \in N'B$. Since $\kappa^{e_2}(x)$ has been proved nonempty, $\kappa^{e_0}(x) \cap \kappa^{e_1}(x)$ is.

A similar reasoning applies to any finite intersection of κ^e 's. But for the quotients $e \in \mathcal{E}_B$ in a coherent category \mathbb{B} the compactness applies: if any finite family is consistent, then the whole set together is. Therefore, $\kappa B(x)$ is nonempty.

(iii) So we can surely chose $kB(x) \in \kappa B(x)$. No matter which element we choose, the equation

$$N''e \circ kB = kFA \circ N'e \quad (10)$$

will hold for every $e \in \mathcal{E}_B$, because $kFA = hA$ and the definition of κB implies

$$N''e \circ \kappa B = hA \circ N'e$$

On the other hand, recall that $N''B = \Phi M''B$ was defined so as to make the function $\langle N''e \rangle_{e \in \mathcal{E}_B}$ injective. But this means that the set of equations (10), for all $e \in \mathcal{E}_B$, together determine at most one $kB(x)$, since the functions $N''e$ are jointly injective.

So the family $hA : N'FA \rightarrow N''FA$, $A \in \mathbb{A}$, extends to a uniquely determined family $kB : N'B \rightarrow N''B$, $B \in \mathbb{B}$.

(iv) To prove that the family $kB : N'B \rightarrow N''B$ is natural, take an arbitrary arrow $g : B_0 \rightarrow B_1$ in \mathbb{B} and an arbitrary $e_1 : B_1 \rightarrow FA_1$ from \mathcal{E}_{B_1} . Let e_0 be the coimage of $e_1 \circ g$

$$\begin{array}{ccc} B_0 & \xrightarrow{g} & B_1 \\ e_0 \downarrow & & \downarrow e_1 \\ FA_0 & \xrightarrow[d]{} & FA_1 \end{array} \quad (11)$$

The codomain of e_0 is in the image of F because it is powerful.

We want to prove that the upper square in the diagram

$$\begin{array}{ccccc} N'B_0 & \xrightarrow{kB_0} & N''B_0 & & \\ N'g \downarrow & & \downarrow N''g & & \\ N'B_1 & \xrightarrow{kB_1} & N''B_1 & & \\ N'e_1 \downarrow & & N''e_1 \downarrow & & \\ N'FA_1 & \xrightarrow[hA_1]{} & N''FA_1 & & \\ N'FA_0 & \xrightarrow[hA_0]{} & N''FA_0 & & \\ N'd \swarrow & & \searrow N''d & & \\ & & & & \end{array}$$

commutes. The lower square and the large outside trapezoid surely commute by the definition of kB . The small trapezoid commutes by the naturality of h , and the two triangles simply as the images of (11). Chasing, one concludes that

$$N''e_1 \circ kB_1 \circ N'g = N''e_1 \circ N''g \circ kB_0$$

But e_1 was taken as an arbitrary element of \mathcal{E}_{B_1} , so the last equation holds for all such. Since they are, by the construction of $N'' = \Phi M''$, jointly monic,

$$kB_1 \circ N'g = N''g \circ kB_0$$

follows.

This completes the proof of $(b) \Rightarrow (c)$. The converse $(c) \Rightarrow (b)$ can be proved by modifying [22, thm. 7.1.4–4']. The argument is lengthy, based on the Los-Tarski theorem, and I do not see a way to improve on it, so the reader may wish to consult the original.

Finally, to connect (d) with the other three conditions, note that the assumption of coproducts makes $\text{Mod}(\mathbb{B})$ into a locally finitely presentable category, so that $F^\#$ must have a left adjoint, like in [10, § 5], obtained by restricting the left Kan extension of F . Hence $(d) \Leftrightarrow (a)$. But a proof of this was already in [11] and [15], albeit in a slightly different setting. \square

An immediate consequence of theorem 1 and proposition 4 is a precise syntactic characterisation of *definitional extensions*, the interpretations F which induce an equivalence $F^\#$ between the model categories. The class is essentially larger than assumed in any of the implemented versions.

Corollary 1. $F^\# : \text{Mod}(\mathbb{B}) \rightarrow \text{Mod}(\mathbb{A})$ is an equivalence if and only if $F : \mathbb{A} \rightarrow \mathbb{B}$ is a powerful embedding, and subcovering.

A proof of this can also be derived from Makkai-Reyes' *conceptual completeness* theorem [22, thm. 7.1.8], which is the main result of their book.

4.5. Conclusions and further work

The research reported in this paper was originally motivated by the questions arising from the semantics and the usage of SPECWARE™, a tool for the automatic synthesis of software systems, developed at Kestrel Institute. In particular, the original semantics of pspecs as an abstract family of arrows [29] needed to be refined into a precise syntactic characterisation and verified semantically. This task took us far afield, into nontrivial model theory and functorial calculus, and brought about the above theorem relating two extant notions of parametricity. As suggested at the end of section 4.2.4., it can be viewed as an indexed completeness result. Formalizing this view might lead to various conceptual and meta-theoretical insights.

But the question of the practical repercussions of the presented material, or of their absence, seems even more interesting. The immediate task should probably be to analyze closely related families of coherent functors, capturing the instantiations and the implementations of theories. The practice of parametric specification is based

upon them as much as upon the family of pspecs, studied in the present paper. Some important issues of refinement directly require this further analysis.

However, as we are not very far in any of these tasks, the main point of presenting this work currently is not this or that particular result, but showing categorical model theory at work in the software specification framework and suggesting a first step or two toward developing specific tools for analyzing and designing specification frameworks.

If, as is often stated, the increasing complexities and dynamics of evolving software development tasks make semantical analyses increasingly important, even indispensable in critical cases, then mathematical methods of the kind presented here may come to play an increasingly important role, as they may provide enough abstraction to resolve the concrete problems where formal methods are genuinely needed.

5. Other mathematical methods

In this section, we present three methods for structuring theories: limits, interpretations, and slicing. Limits allow us to find the “semantic intersection” of a collection of theories, interpretations allow us to relate specifications in a more general way, and slicing allows us to split a theory into a collection of meaningful parts.

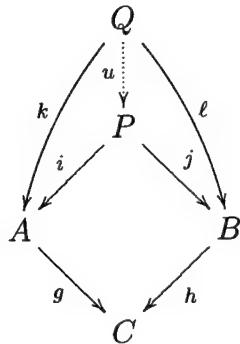
5.1. Limits

The category of specifications in Specware has all finite colimits. Not only are they useful, but they are efficiently computable with a linear time algorithm. Do limits of specifications exist? Would they be useful? Can they be computed efficiently when they exist? As a special case, what is the product of two specifications? what it would mean? how could it be used? Some potential uses of limit computations include: slicing, evolution, filling out a partial parallel refinement.

There is a general result about categories that provides a remarkable inductive computation of limits provided certain basic special cases exist.

Proposition 1. If a category has a final object and all pullbacks, then it has all finite limits.

Proof: It is fairly easy to show that if a category has a final object, products, and equalizers then it has all limits [4]. It can be further shown that if a final object and pullbacks exist, then we can compute products and equalizers; e.g. the product of two objects A and B is the pullback of the diagram $A \longrightarrow \mathbf{1} \longleftarrow B$ where $\mathbf{1}$ is the final object.



The diagram on the left gives a pullback situation. Given specs A , B , and C and morphisms $A \xrightarrow{g} C \xleftarrow{h} B$, does

a cone $A \xleftarrow{i} P \xrightarrow{j} B$ exist such that

- (1) the diagram commutes: $g \circ i = h \circ j$ and
- (2) the cone is universal: for any other cone $A \xleftarrow{k} Q \xrightarrow{\ell} B$ (such that $g \circ k = h \circ \ell$) there exists a unique arrow $u : Q \rightarrow P$ that factors k and ℓ ; i.e. such that $k = i \circ u$ and $\ell = j \circ u$. ^a

^a The definition of cone includes commuting of diagram.

We approach the question of whether limits of specifications exist by examining incrementally the following categories of interest:

SIG = Category of signatures and their morphisms; A signature consists of a set of sort symbols and a set of (higher-order) operator symbols together with their arities. A signature morphism maps each symbol of the domain signature to a symbol of the codomain, such that the arities of each operator symbol is preserved under translation. For example, suppose that the domain has sorts D , and R and operator f , and the codomain has sorts E , and S and operator g . The map m is a signature morphism: $m = \{f \mapsto g, D \mapsto E, R \mapsto S\}$. Note that m ensures compatible translation of the arity of f . To put it another way, a signature morphism is required to preserve the sort constructions of a signature, whether these constructions arise in giving the arity of an operator or in explicit sort definitions.

A signature morphism translates an expression, such as an axiom, by context-free translation of constituent symbols.

For simplicity we assume that the arity of an operator is built up from products and function (exponentials). We assume that the boolean sort *boolean* and the logical quantifiers and operators $\forall, \exists, \wedge, \vee, \neg, \Rightarrow, \Leftarrow$ are built in. All sorts are equipped with an equality.

THY = Category of theories and their morphisms; i.e. an object in THY is a signature plus a set of sentences that are closed under entailment (called the *theorems* of the theory). The morphisms are essentially signature morphisms, with the additional condition that they translate theorems to theorems.

SPEC = the category of (higher-order) specs and their morphisms. An object in SPEC is a signature plus a finite (or more generally recursive) set of sentences called the *axioms* of the specification. The morphisms are essentially signature morphisms, with the additional condition that they translate axioms to theorems. This condition implies that the morphisms also translate theorems to theorems.

Say something about other sort constructions such as sums, quotients, and subsorts? Same general idea, but the final object is just more complex?

Proposition 2. SIG has all finite limits.

Proof: We need to show that SIG has a final object and pullbacks. The final object $\mathbf{1}_{SIG}$ consists of one sort (say D) and one operator for each finite arity (i.e. $f_{m,n} : D^m \rightarrow D^n$ where $m \geq 0 \wedge n \geq 0$). To show universality, note that there is a unique arrow from an arbitrary signature $S \rightarrow \mathbf{1}_{SIG}$ which takes each sort to D and each operator of S to the unique operator in $\mathbf{1}_{SIG}$ that guarantees signature compatibility.

The pullback of signatures is the fiber product over C of the sorts and ops. If γ is a sort of C , then the fiber over γ is $g^{-1}(\gamma) \times h^{-1}(\gamma)$. The elements of this fiber are pairs of sorts, say $\langle \alpha, \beta \rangle$, which can be thought of as the sort product $\alpha \times \beta$. If c is an operator of C , then the fiber over c is $g^{-1}(c) \times h^{-1}(c)$. The elements of this fiber are pairs of operators, say $\langle a : D \rightarrow R, b : E \rightarrow S \rangle$, which can be thought of as the function product $a \times b : D \times E \rightarrow R \times S$ such that $a \times b : \langle d, e \rangle \mapsto \langle f(d), g(e) \rangle$.

To show universality, let $A \xleftarrow{k} Q \xrightarrow{\ell} B$ be another cone. Define the universal arrow $u : Q \rightarrow P$ as follows. For sort σ in Q , let $u : \sigma \mapsto \alpha \times \beta$ if $k(q) = \alpha$ and $\ell(q) = \beta$. For each operator q of Q , let $u : q \mapsto a \times b$ if $k(q) = a$ and $\ell(q) = b$. This is unique since no other translation will commute. QED

The pullback in SIG can be easily computed in time that is $O(\max(|A|, |B|, |P|))$ (which is at most $|A| \times |B|$).

Next we look at limits in the category of theories, THY. The key idea here is that a theorem in the pullback corresponds to theorems in the components A and B that have the same abstract form as some theorem in C . (clarify!)

Proposition 3. THY has all finite limits.⁶

Proof: We need to show that THY has a final object and pullbacks. The final object in THY is essentially the final object of SIG $\mathbf{1}_{SIG}$ together with the set of all sentences constructable in that language. Naturally it is inconsistent.

To show universality, simply note that there is a unique arrow from the signature of an arbitrary theory $T \rightarrow \mathbf{1}_{THY}$ and that it translates each source theorem to the unique theorem in $\mathbf{1}_{THY}$ that has the same abstract form.

The pullback of theories extends the pullback on the underlying signatures. The fiber over a theorem t of C is essentially $g^{-1}(t) \times h^{-1}(t)$. Each theorem in the fiber corresponds to a pair $\langle t_A, t_B \rangle$ and can be represented in the language of P as follows. First, note that t , t_A , and t_B have the same abstract form because of the context-free way that the underlying signature morphisms translate expressions. We define a function that recursively translates a pair of expressions into the language of P :

$$\begin{aligned} \text{Translate}(\forall(x : s_A)P_A(x), \forall(x : s_B)P_B(x)) &= \forall(x : s_A \times s_B)\text{Translate}(P_A(x), P_B(x)) \\ \text{Translate}(P_A \wedge Q_A, P_B \wedge Q_B) &= \text{Translate}(P_A(x), P_B(x)) \wedge \text{Translate}(Q_A(x), Q_B(x)) \end{aligned}$$

⁶ Should the pullback in THY include all defined ops? What if the pullback in SIG is empty, yet the pullback in THY could have lots of paired theorems if the right ops were there...

$\text{Translate}(f_A(a_A), f_B(a_B)) = f_A \times f_B(\text{Translate}(a_A, a_B))$

$\text{Translate}(c_A, c_B) = c_A \times c_B$ (for constants c_A and c_B)

and so on. The existence of the sort $s_A \times s_B$ in P is guaranteed by the context-free translation of expressions by signature morphisms and by the assumption that $g(t_A) = t = h(t_B)$. Similar guarantees apply to $f_A \times f_B$, $c_A \times c_B$, etc.

We need to show that this object is a theory; i.e. it is closed under entailment (or logical consequence). A sketch:⁷ Consider a set of theorems of P that are formed as defined above, call them $\{t_{P_e}\}_{e=1,\dots,n}$. By construction we have

$$\begin{array}{ccc} & t_{P_e} & \\ i \swarrow & & \searrow j \\ t_{A_e} & & t_{B_e} \\ g \searrow & & h \swarrow \\ & t_{C_e} & \end{array}$$

for $e = 1, \dots, n$. Suppose that we can infer t_P from $\{t_{P_e}\}_{e=1,\dots,n}$ via rule R . Under the mild assumption that R acts on the syntactic form of theorems, then the analogous inferences will be made in A , B , and C , so we have

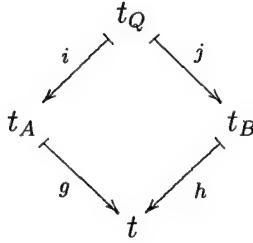
$$\begin{array}{ccc} & t_P & \\ i \swarrow & & \searrow j \\ t_A & & t_B \\ g \searrow & & h \swarrow \\ & t_C & \end{array}$$

i.e. inductively we have t_A , t_B , and t_C as theorems of A , B , and C respectively, so t_P is a theorem of P by construction. Consequently, P is closed under inference and it forms a theory.

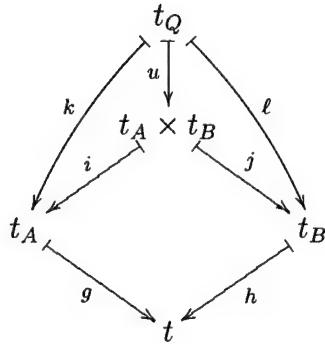
We also need to show that i and j are theory morphisms and that the square commutes. The projection morphisms i and j simply unpack the theorem $t_A \times t_B$ into t_A and t_B respectively (from whence it was formed). The diagrams above indicate the essential reason for the commuting of the diagram.

To show universality, define the universal arrow $u : Q \rightarrow P$ as we did in SIG. We must show that it translates theorems to theorems and that it factors k and ℓ . Suppose that

⁷ We do induction on the proof structure of an arbitrary theorem in P .



then



shows the action of u and the factoring of k and ℓ with respect to theorems. QED

Corollary 1. The product of two theories exists and is comprised of theorems that have the same abstract form in both A and B (clarify!).

Next we look at limits in the category of specs, SPEC. Unfortunately these do not in general exist. For example, The final object in SPEC, $\mathbf{1}_{SPEC}$, must be a finite (or recursive) presentation of $\mathbf{1}_{THY}$. One can imagine some meta-machinery for presenting this finitely, such as an algorithm for enumerating it, but this is not a straightforward presentation format.

A more difficult question is the existence of a finite/recursive axiomatization of a pullback theory. If we just consider the pullback on the axioms of A and B , there may be none with the same arity-structure, so the fiber product of theorems would be empty (which is not a sufficient axiomatization!).

A special case: if we consider the product of a spec with itself (e.g. the product of group theory), then we do get an axiomatization of the product, which is isomorphic to the spec itself (so the product of groups is a group!).

Nevertheless, pullbacks do exist for a wide subcategory of SPEC. Monics in SPEC correspond exactly to morphisms in which the underlying signature morphism is injective and axioms translate to axioms (?). Monics in SPEC include identity arrows, translate arrows, definitional extensions, c-def arrows, imports/inclusions/extensions, conservative arrows, and compositions of these.

Proposition 4. Pullbacks of monics exist in SPEC.

Proof: The key idea is that the fiber over a symbol or axiom is a singleton set. We show this result for simple inclusions, but the generalization to arbitrary monics is straightforward. Suppose that g and h are inclusions, then $\text{symbols}(P) = \text{symbols}(A) \cap \text{symbols}(B)$ and $\text{axioms}(P) = \text{axioms}(A) \cap \text{axioms}(B)$. Note that each symbol in the axioms of P must be symbol in both A and B , so the presentation of P is closed⁸. Next note that P is included in A and in B , so we have a cone.

To show universality, let $A \xleftarrow{k} Q \xrightarrow{\ell} B$ be another cone. Since Q is included in A and in B , it must be included in $A \cap B$ which is P . Such an inclusion is unique. QED

Proposition 5. Pullbacks (when they exist) preserve identity arrows, translate arrows, definitional extensions, c-def arrows, imports/inclusions/extensions, conservative arrows, and monics, and p-spec arrows.

5.2. Interpretations

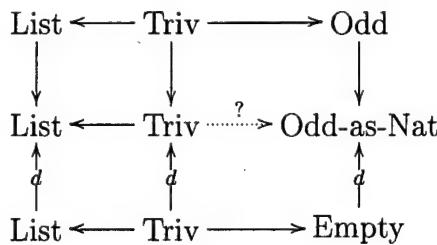
In the 1970's, Goguen and Burstall discovered the use of pushouts to instantiate parametrized specifications. In Specware, we tried to generalize this idea and use pushouts to instantiate *parametrized interpretations*; however, it didn't seem to work. We named this problem "triv-to-triv-via-subsort", after one of the proposed but unsatisfactory solutions.

This section proposes to solve this problem by recasting interpretation morphisms as squares of interpretations rather than triples of morphisms.

As an example, let's try to instantiate List-to-List (the identity) with Odd-to-Nat. We expect to obtain an interpretation from List[Odd] to List[Nat]. For reference, here's the mediator of Odd-to-Nat:

```
spec Odd-as-Nat is
    import Empty
    sort Odd = Nat | odd?
end-spec
```

We try to take pushouts according to this diagram:



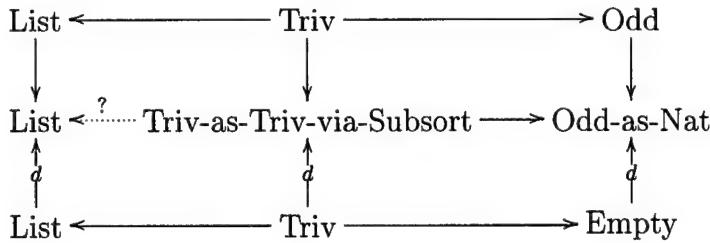
⁸ extra constraint on a spec?

However, we cannot define a suitable morphism from Triv to Odd-as-Nat. If we map E to Odd then the upper square commutes. If we map E to Nat then the lower square commutes. We can't have both.

To fix the problem, we tried replacing Triv-to-Triv with Triv-to-Triv-via-Subsort:

```
spec TRIV-as-TRIV-via-SUBSORT is
    sorts E-dom, E-cod
    sort E-dom = E-cod | p?
    op p? : E-cod -> Boolean
end-spec
```

We can find an interpretation morphism from Triv-to-Triv-via-Subsort to Odd-to-Nat, but not to List-to-List:



So this idea pushes the problem around but doesn't solve it.

In the example above, we tried to construct an interpretation from the List[Odd] to List[Nat]. Although this seems reasonable, it is not. To see why, let's try a similar example that makes the error more apparent. Instead of Odd-to-Nat, let's try Pair-to-Nat, which refines an abstract sort Pair to a pair of naturals:

```
spec Pair-as-Nat is
    import Empty
    sort Pair = Nat, Nat
end-spec
```

If the instantiation were to succeed, we would obtain a refinement from List[Pair] to List[Nat]. What would this refinement look like? We could keep two lists of naturals, one for the left component, one for the right. Such a construction depends on an injection:

$$List(Pair(A)) \rightarrow Pair(List(A))$$

However, this is not what we want. We want to refine List[Pair] to List[Nat × Nat], not List[Nat].

It turns out that we can obtain the desired result by replacing interpretation morphisms with commuting squares of interpretations. Thus:

$$\begin{array}{ccc} A & \xrightarrow{\quad\quad\quad} & C \\ \Downarrow & & \Downarrow \\ B & & D \end{array} \quad \text{becomes} \quad \begin{array}{ccc} A & \xrightarrow{\quad\quad\quad} & C \\ \Downarrow & & \Downarrow \\ B & \xrightarrow{\quad\quad\quad} & D \end{array}$$

Then there is no problem constructing the appropriate diagram:

$$\begin{array}{ccccc} & & \text{List} & \leftarrow & \text{Triv} & \longrightarrow & \text{Pair} \\ & & \Downarrow & & \Downarrow & & \Downarrow \\ & & \text{List} & \leftarrow & \text{Triv} & \longrightarrow & \text{Empty} \end{array}$$

The interpretation from Pair to Empty constructs pairs of naturals, as does the interpretation from Triv to Empty; thus, the right square commutes. The upper pushout constructs List[Pair] and the lower one constructs List[Nat × Nat], as desired.

5.3. Slicing

The term “theory slicing” refers to two different problems:

- To factor specifications into pieces that are likely to be reusable.
- To eliminate operators and axioms that not needed for a given set of operator definitions.

We only discuss the second kind here; its primary use is in code generation to minimize the size of the target code.

We structure the relationship between sets of operations and axioms using Galois connections.

A *partial order* is a set A with a relation \leq that is reflexive, transitive, and antisymmetric. Antisymmetric means that $a \leq b$ and $b \leq a$ implies $a = b$.

A *Galois connection* or *Galois pair* (F, G) is a pair of monotone functions

$$\begin{aligned} F : A &\rightarrow B \\ G : B &\rightarrow A \end{aligned}$$

between partial orders A and B such that $a \leq GFa$ and $FGb \leq b$.

An isomorphism is a Galois connection in which $a = GFa$ and $FGb = b$. Unlike an isomorphism, a Galois connection is asymmetric: (G, F) may not be Galois even if (F, G) is. The categorical concept of adjunction further generalizes a Galois pair, but we don't need adjunctions here.

Given a spec S , let $Axiom$ be the set of all axioms in S and let Op be the set of all operators. We define $Axioms = \mathcal{P} Axiom$ and $Ops = \mathcal{P} Op$. That is, $Axioms$ and Ops are the sets of all subsets of axioms and operations. That is, an element of $Axioms$ is a set of axioms. The sets $Axioms$ and Ops are preorders, ordered by set containment.

We can define functions

$$\begin{aligned} F : Axioms &\rightarrow Ops \\ G : Ops &\rightarrow Axioms \end{aligned}$$

so that F maps a set of axioms to the set of operations used by the axioms, and G maps a set of operators to the set of axioms that use at most these operations. F is monotone because more axioms use more operators, and G is monotone because more operations allow us to state more axioms.

The function GF takes a set of axioms a to the (larger) set of axioms stateable using the operators of a . The function FG takes a set of operators b to the (smaller) set of operators that actually occur in the axioms stateable using b . Thus (F, G) is a Galois pair.

An equivalent formulation of Galois pair is that for all a and b ,

$$Fa \leq b \iff a \leq Gb$$

and it is worth checking that this condition also holds in our example.

For any Galois connection, we can show that the composites FG and GF are closure operators, that is, that they are idempotent, that is, that $FGFG = FG$ and $GFGF = GF$. Thus, we only need to apply these operators once; applying them further has no effect. Here is the idempotence proof for GF :

$$\begin{aligned} a &\leq GFa && \text{assumption} \\ GFa &\leq GFGFa && \text{monotonicity of } GF \\ FGb &\leq b && \text{assumption} \\ FGFa &\leq Fa && \text{substitution of } Fa \text{ for } b \\ GFGFa &\leq GFa && \text{monotonicity of } G \\ GFGFa &= GFa && \text{antisymmetry of } \leq \end{aligned}$$

A *partial order with complement* is a partial order A with an operation $_^c : A \rightarrow A$ such that

$$\begin{aligned} (a^c)^c &= a && \text{and} \\ a \leq b &\iff b^c \leq a^c \end{aligned}$$

Given a monotone function $F : A \rightarrow B$, we can define $F^c : A \rightarrow B$ by $F^c a = F(a^c)^c$. Then, if (F, G) is a Galois pair, so is (F^c, G^c) :

$$\begin{aligned}
& G^c b \leq a \\
\iff & G(b^c)^c \leq a \quad \text{definition of } G^c \\
\iff & a^c \leq G(b^c) \quad \text{complement reverses order} \\
\iff & F(a^c) \leq b^c \quad F \text{ and } G \text{ are Galois} \\
\iff & (b^c)^c \leq F(a^c)^c \quad \text{complement reverses order} \\
\iff & b \leq F(a^c)^c \quad \text{complement is an involution} \\
\iff & b \leq F^c a \quad \text{definition of } F^c
\end{aligned}$$

In our example, $F^c a$ is the set of operations that don't occur outside a , and $G^c b$ is the set of axioms that use some operation from b . F^c is hard to grasp, but G^c is quite natural.

The map $G^c F^c$ takes a set of axioms a to the subset obtained by throwing out axioms all of whose operations occur outside a . The map $F^c G^c$ takes a set of operations b and enlarges it by examining the axioms that don't touch b and adding the other operations they don't use. As before, both these maps are idempotent.

There is another pair of Galois connections available to us. For the moment, we define a *specification* to be a pair (O, A) of operations and axioms such that $\text{ops } A \subseteq O$. Then the subspecifications of a spec S form a partial order with complement \mathcal{PS} under pairwise containment.

We define four monotone functions from $\mathcal{P}(S)$ to itself:

$$\begin{aligned}
Ms &= \text{remove all operations not used in axioms} \\
Ns &= \text{add all operations of } S \text{ to } s \\
Is &= \text{remove all axioms of } s \\
Js &= \text{add all axioms stateable using operations of } s
\end{aligned}$$

Then (M, N) and (I, J) are both Galois.

References

1. J. Adámek, P.T. Johnstone, J.A. Makowsky, and J. Rosický. Finitary sketches. *Journal of Symbolic Logic*, 62:699–707, 1997.
2. J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*, volume 189 of *LMS Lecture Notes*. Cambridge University Press, 1994.
3. James Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
4. Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, Englewood Cliffs, NJ, 1990.
5. Lee Blaine, Limei Gilham, Junbo Liu, Douglas Smith, , and Stephen Westfold. Planware – domain-specific synthesis of high-performance schedulers. In *Proceedings of the Thirteenth Automated Software Engineering Conference*, pages 270–280. IEEE Computer Society Press, October 1998.
6. R. M. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In D. Bjørner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*). Springer LNCS 86, 1980.

7. Th. Dimitrakos. *Formal Support for Specification Design and Implementation*. PhD thesis, University of London, 1998.
8. H. Ehrig and M. Große-Rhode. Functorial theory of parametrized specifications in a generalized specification framework. *Theoretical Computer Science*, 135:221–266, 1994.
9. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs in Theoretical Computer Science*. Springer, 1990.
10. P. Gabriel and F. Ulmer. *Lokal Präsentierbare Kategorien*, volume 221 of *Lecture Notes in Mathematics*. Springer, 1971.
11. H. Ganzinger. Parametric specifications: parameter passing and implementations with respect to observability. *ACM Transactions on Programming Languages and Systems*, 5:318–354, 1983.
12. J. Goguen. Parametrized programming. *Transactions on Software Engineering*, 10(5):528–543, 1984.
13. J. Goguen. *Principles of parametrized programming*, pages 159–225. Addison-Wesley, 1989.
14. J. Goguen and R.M. Burstall. Cat: a system for the structured elaboration of correct programs from structured specifications. Technical Report CSL 118, SRI, 1980.
15. J. Goguen and J. Meseguer. Universal realization, persistent interconnection, and implementation in abstract modules. In *Proceedings of the Ninth ICALP, LNCS 140*, pages 265–281. Springer, 1982.
16. Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence*, pages 219–239, 1969.
17. Kestrel Institute. *Specware Language Manual*, 1998. Available from keep@kestrel.edu.
18. S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 1971.
19. F.W. Lawvere. *Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963.
20. M. Makkai. Stone duality for first order logic. *Advances in Math*, 65, 1987.
21. M. Makkai and R. Paré. *Accessible Categories: The Foundations of Categorical Model Theory*, volume 104 of *Contemporary Mathematics*. American Mathematical Society, 1989.
22. M. Makkai and G. Reyes. *First Order Categorical Logic*, volume 611 of *Lecture Notes in Mathematics*. Springer, 1977.
23. E. Manes. *Algebraic Theories*. Springer, 1976.
24. J. Meseguer. General logics. In H. Ebbinghaus, editor, *Logic Colloquium 87*, pages 275–329. North Holland, Amsterdam, 1989.
25. Douglas R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming*, 15(5-6):571–606, May-June 1993.
26. Douglas R. Smith. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96*, volume LNCS 1101, pages 62–84. Springer, 1996.
27. Douglas R. Smith. Mechanizing the development of software. In M. Broy and R. Steinbrueggen, editors, *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, pages 251–292. IOS Press, Amsterdam, 1999.
28. Y. V. Srinivas and Richard Jüllig. Specware: Formal support for composing software. In B. Moeller, editor, *Proceedings of the Conference on Mathematics of Program Construction LNCS 947*, pages 399–422. Springer, Berlin, 1995.
29. Y.V. Srinivas. Refinement of parametrized algebraic specifications. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 164–186. Chapman & Hall, 1997.
30. C. Strachey. Fundamental concepts in programming languages. Unpublished lecture notes, 1967.

DISTRIBUTION LIST

addresses	number of copies
CRAIG S. ANKEN AFRL/IFTD 525 BROOKS ROAD ROME, NY 13441-4505	5
KESTREL INSTITUTE 3260 HILLVIEW AVENUE PALO ALTO, CA 94304	1
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-5218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
SOFTWARE ENGR'G INST TECH LIBRARY ATTN: MR DENNIS SMITH CARNEGIE MELLON UNIVERSITY PITTSBURGH PA 15213-3890	1
AFIT/ENG ATTN: TOM HARTRUM WPAFB OH 45433-6583	1
SOFTWARE ENGINEERING INSTITUTE ATTN: MR. WILLIAM E. HEFLEY CARNEGIE-MELLON UNIVERSITY 304 OAK GROVE CT WESFORD PA 15090	1

AFIT/ENG 1
ATTN: DR GARY B. LAMONT
SCHOOL OF ENGINEERING
DEPT ELECTRICAL & COMPUTER ENGRG
WPAFB OH 45433-6583

NSA/OFC OF RESEARCH 1
ATTN: MS MARY ANNE OVERMAN
9800 SAVAGE ROAD
FT GEORGE G. MEADE MD 20755-6000

DARPA/ITO 1
ATTN: DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714

NASA/JOHNSON SPACE CENTER 1
ATTN: CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058

NATIONAL INSTITUTE OF STANDARDS 1
AND TECHNOLOGY
ATTN: CHRIS DABROWSKI
ROOM A266, BLDG 225
GAITHSBURG MD 20899

EXPERT SYSTEMS LABORATORY 1
ATTN: STEVEN H. SCHWARTZ
NYNEX SCIENCE & TECHNOLOGY
500 WESTCHESTER AVENUE
WHITE PLAINS NY 20604

NAVAL TRAINING SYSTEMS CENTER 1
ATTN: ROBERT BREAUX/CODE 252
12350 RESEARCH PARKWAY
ORLANDO FL 32826-3224

DR JOHN SALASIN 1
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DR BARRY BOEHM 1
DIR, USC CENTER FOR SW ENGINEERING
COMPUTER SCIENCE DEPT
UNIV OF SOUTHERN CALIFORNIA
LOS ANGELES CA 90089-0781

DR STEVE CROSS
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891

1

DR. DAVE GUNNING
DARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

1

SPAWARSCEN D44209
ATTN: LEAH WONG
53245 PATTERSON ROAD
SAN DIEGO, CA 92152-7151

1

SPAWARSCEN D4123
ATTN: LES ANDERSON
53560 HULL STREET
SAN DIEGO CA 92152-5001

1

DIRNSA
ATTN: MICHAEL R. WARE
DOD, NSA/CSS (R23)
FT. GEORGE G. MEADE MD 20755-6000

1

INSTITUTE OF TECH DEPT OF COMP SCI
ATTN: DR. JAIDEEP SRIVASTAVA
4-192 EE/CS
200 UNION ST SE
MINNEAPOLIS, MN 55455

1

AFRL/IFT
525 BROOKS ROAD
ROME, NY 13441-4505

1

AFRL/IFTM
525 BROOKS ROAD
ROME, NY 13441-4505

1

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.